Design of Information Channels for Mission-Critical Systems

by

Brian A. Purville

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 23, 2001

Author_____

Department of Electrical Engineering and Computer Science
May 23, 2001

Certified
by_____

Dr. Amar Gupta
Thesis Supervisor

Accepted
by_____

Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Design of Information Channels for Mission Critical Systems
by
Brian A. Purville

Submitted to the
Department of Electrical Engineering and Computer Science

May 23, 2001

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Current mission critical systems are distributed client-server applications written in C/C++, designed to work for a specific hardware or software vendor. These systems generally are not portable and do not permit the easy addition and modification of modules. The vision for these mission critical systems is a web/browser-enabled system that is vendor and operating system independent, and allows the easy addition and modification of system components. This paper examines using the Joint Battlespace Infosphere (JBI) to achieve this vision. The JBI is a distributed component architecture that uses the publish/subscribe paradigm. It is unique in that it makes the flow of information through the system explicit by expressing the system as a set of information channels from the data sources to clients. The results showed that the JBI can be used to successfully "web-enable" current mission critical systems.

Thesis Supervisor: Dr. Amar Gupta
Title: Co-Director, Productivity From Information Technology Initiative Sloan School of Management

Table of Contents

Chapter 1. Introduction

The World Wide Web (WWW) can be dated back to 1989 when Tim Berners-Lee wrote the original WWW proposal "Information Management: A proposal" [3]. The proposal concerned the management of general information about accelerators and experiments at CERN, the European organization for nuclear research. The hypertext, browsing, and linked information systems Berners-Lee talked about in his paper was the basis of the WWW. The WWW has come a long way since then. Originally designed just for use in nuclear research, it has grown to affect millions of people in diverse fields. The Web, along with the Internet, has revolutionized the way we gather, process, and use information. It has redefined the meaning and processes of business, commerce, marketing, finance, publishing, education, research, and development, along with other aspects of our daily life [19].

Anyone who has used the Internet realizes the benefits it brings. A student sitting in his or her dorm can email her professor the latest version of a paper in seconds instead of walking across the campus to the professor's office. A basketball fan connected to the Internet can find out who won the big game almost immediately rather than having to wait for the television news to air or the local newspaper to arrive. A credit card bill can even be paid with just one click of a mouse button. These are just a few of the countless ways in which the Internet and the WWW have made people's lives easier. All the examples so far relate to the user experience, but the Internet has also benefited software developers. Emerging Internet technologies such as Java and the Extensible Markup Language (XML) have made the design and implementation of large distributed

applications easier. Through the use of a web browser, an application developer can write a program, deploy it on a server, and have it used by thousands of users across the world. There is no need to install the application on the computers of every user. The Java programming language has allowed the implementation of applications that can be ported to different operating systems with little to no work. Java also comes with a large set of application programming interfaces (APIs) that facilitate the design and implementation of large distributed systems. XML allows distributed applications to define a data schema with which to transmit information and a format to read and write documents that conform to the defined schema. XML and Java, along with the many tools that have been developed to work with them, expedite the process of designing a distributed system in which the data needs to be delivered to many users.

Due to the overwhelming increase in popularity of the Internet and the WWW and the major benefits they bring, there has been an increased interest in the research of Internet technologies. Techniques for effectively applying the Internet and its associated technologies are being studied and developed. There are several major conferences, journals, and workshops devoted to web and Internet research such as the International Conference on Web Information Systems Engineering, the International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems, and IEEE Internet Computing. Zhong, Liu, Yao, and Ohsuga went as far as to define a new research field named Web Intelligence (WI) [19] for the systematic study of advanced Web technology and developing Web-based intelligent information systems. The WI

related topics include Web Information Management, Web Information Retrieval, Web-Based Applications, and Web Information System Environment and Foundation.

The commercial world has embraced the WWW and the Internet. Now government agencies wish to follow suit. Currently, many government systems are based on the older client-server approach. This approach differs from the newer web approach in several ways. Many of the old client server programs are written in C and C++, rather than in Java. Though C/C++ programs outperform Java in terms of speed, Java programs are better in terms of portability and maintainability. Another difference between the two approaches is that the client-server approach often involves fat clients (client application) while the web approach involves thin clients (web browser). When the network supports the paradigm, thin clients are preferable to fat clients, because software updates can be made at the server and the thin client need not be changed. In a client-server architecture, software updates often require modifying all the fat clients that have been deployed. Transforming these government client-server systems into web-based systems could greatly reduce maintenance and development costs, while adding functionality and usability.

One system that the U.S. Government desires to "web-enable" is the Theater Battle Management Core System (TBMCS). TBMCS is the mission-critical system used to plan and manage an overall air campaign as well as to plan and execute the daily air war. It is a client-server architecture in which clients written in C++ make calls to database servers. A central function of TBMCS is to move the data from the database servers to

the clients quickly and reliably. Interested clients should be informed of data changes as soon as possible. The path through which information flows from the database to the client can be thought of as an information channel. TBMCS requires that these channels be customized for different users. For example, the level of detail a soldier in the field wants may be different than what a commander wants. Also, even if two users want the same information, the way they want it displayed may be different. One user may be accessing the application from a WAP-enabled handheld device while another may be sitting in front of a PC. The information needs to be sent to both users in the right format. TBMCS needs to be able to support all these users with different needs and preferences.

Consistent with the government's goal of web-enabling their combat systems, they are developing an architecture called the Joint Battlespace Infosphere (JBI). The JBI is a platform that uses the publish/subscribe paradigm to support multiple data producers and data consumers. An example of a producer is a satellite system that sends out satellite data. A possible consumer could be a soldier sitting in front of a PC viewing the satellite data. As the information flows from the producer to the consumer, it passes through components called fuselets that manipulate the data. The JBI integrates many individual information systems and acts as an intermediary between these systems. It also tailors the information for individual users: the commander gets a high level view of the military campaign, while the soldier in the field gets a detailed description of a nearby hostile base. The JBI platform is web-enabled and uses cutting edge web technologies such as SOAP XML for remote procedure calls over HTTP [27]. Many of the requirements of

TBMCS appear to be satisfied with the JBI. If one views the TBMCS database servers as producers in the JBI, the JBI architecture can be used to channel information from the databases to the clients in the system.

This paper describes the author's efforts to enable information channels in older client-server mission-critical systems such as TBMCS by redesigning them to use web technologies such as Java and XML. The JBI was chosen as the platform on which to create the information channels. Using the architecture of the JBI, TBMCS data sources were represented as producers and information was processed by JBI fuselets. Since the details of TBMCS are proprietary, TBMCS producers and consumers were modeled using a fictitious mission-critical system based on TBMCS. A car service system was the model devised for prototyping. The car service system can be readily mapped back to the TBMCS system. Though the focus of the design dealt with TBMCS, the results can be applied to any similar system. Accordingly, the results of the prototype show that the JBI can be used to successfully implement a web-enabled distributed component mission-critical system.

The road map for the rest of this paper is organized as follows. Chapter 2 provides the motivation for the work presented in this paper. The current status of mission critical systems is described along with a vision for future systems. The reason the JBI was chosen is also explained here. Background information about TBMCS and Internet technologies used in the design is presented in Chapter 3. Chapter 4 explains the design of the car service (CS) system. Each component of the system is described, as well as

how the components work together to satisfy the system requirements. The prototype of the system is described in Chapter 5. A subset of the components in the design was chosen for the prototype. The implementation of each of these components, along with their interaction with each other, is explained in this chapter. In Chapter 6, the system is evaluated on its ability to achieve the vision for mission critical systems, performance, and scalability. Areas where further work is suggested are presented in Chapter 7. In Chapter 8, the work of other researchers that relate to the ideas in this paper is described. Finally, Chapter 9 summarizes the results of the research presented in the paper.

Chapter 2. Motivation

In this chapter, motivation for the research is presented. First, the status of mission critical systems is discussed along with a vision for future versions of these systems. Then, several methods for achieving the vision are presented.

Current mission critical systems share many common attributes. Most of them are client-server applications written in C and C++ that are designed to work with a specific hardware and software vendor. Many of these systems have a distributed architecture in which the components are spread across multiple machines. These distributed components work together to achieve a common goal. Often, each of these components is aware of most of the other components in the system and communicates directly with them, resulting in a quadratic number of connections among the components.

These common attributes result in several problems. Applications written in C and C++ are compiled to run on specific hardware and operating system platforms. This makes it difficult to port the applications to different computer systems. A related problem is that the applications are vendor specific. Many mission critical systems are designed to work with specific hardware and software applications. Switching vendors would result in very significant and costly changes. Therefore, it is difficult to take advantage of a new vendor that offers a better product. Finally, the large dependence caused by the quadratic connections among system components makes modifications and additions to the system difficult to implement. A change in a component can potentially cause a

change in all other components in the system. Also, adding a new component may be prohibitive if it needs to coordinate with every other component in the system.

The next generation of mission critical systems need to be web and browser-enabled. A browser-enabled system allows the use of a standard client with a well-known interface, thereby reducing the learning curve for a new user. In addition, all that is required to use a web-enabled system is a web browser. Since web browsers are installed on the vast majority of computers today, the system would be available from numerous locations. Furthermore, a web/browser-enabled system can take advantage of the many technological advances in web technology. In addition to being web enabled, the system should be vendor, hardware, and operating system independent. This system would be easily ported to various machines and be able to work with products from a broad variety of vendors that provide the necessary functionality. Finally, it should be easy to add and modify components to the system. This requires minimizing the dependencies among the components.

A common method to achieve this vision is to use an application server, which is a runtime environment for components written in Java or C/C++. An application server often has a web server based interface. It provides functionality for load sharing, life-cycle control, and redundant components. Other common features include built-in database connectivity, distributed transaction management, and message queuing.

An application server typically resides in the middle tier of a multi-tier application. It receives and processes requests from client applications and communicates with database and legacy systems to respond to the request. In these multi-tier systems, much of the application logic resides in the middle tier. The middle tier often consists of multiple components, possibly distributed, that work together to achieve a goal. There are many commercial application servers available such as WebSphere [24] and WebLogic [22]. Popular component architectures include the Common Object Request Broker Architecture (CORBA) [23] and the Enterprise Java Beans (EJB) architecture [21].

An application server provides a layer of abstraction between the client and the backend systems, such as databases and legacy systems. This reduces the number of connections amongst the components in the system. Rather than a client interfacing with each database to get the information it wants, it only has to interface with the application server, which handles the twin tasks of retrieving information from the database and returning it to the client. Since much of the application logic resides on the server, modifications can be made at the server, and each client will receive the modifications. Often in older designs, the application logic resides on the client and software updates need to be made at each client site. In addition to providing a layer of abstraction, component architectures used in application servers promote reusability, which usually leads to a more modular design with fewer dependencies among the components.

An example of a system that uses an application server is the stock web service system developed at IBM T.J. Watson Research Center [4]. The system uses the WebSphere

application server to receive stock requests, EJBs to represent stock requests, and a DB2 database to store stock requests. The system notifies clients when stocks they are interested in reach a predefined threshold. For example, a client can request to receive an email when the price of an IBM share goes above $100. The system decouples the clients of the stock data from the actual stock data and can scale up to one hundred thousand customers, three million active requests, and over two thousand trades/second.

The application server approach described above is an example of distributed component architecture, if the components that comprise the middle layer are spread across multiple computers. In general, distributed component architectures consist of individual components distributed across multiple machines. Each component performs an individual task to achieve an overall goal. Application servers provide an attractive architectural option to achieve the vision discussed at the beginning of this chapter. They provide most of the functionality for a distributed component system, making it possible for the developer to focus on the application logic.

Hermansson, Johansson, and Lundberg [10] looked at redesigning a monolithic telecommunication system into a distributed component system. By doing so, they reduced the cost of system modifications, increased throughput, and increased scalability, while maintaining the same level of performance. The researchers considered using application servers, but instead chose a proprietary component platform named FDS that is built on top of CORBA. The researchers chose this platform over the application server, because they were more familiar with FDS. They, found however, that the

redesign would have been easier if they had used an application server since it provides functionality that is lacking in the FDS.

The work of Hermansson, Johansson, and Lundberg shows that a component architecture that provides the most functionality should be chosen so that developers can concentrate on developing the components that make up the system. The Joint Battlespace Infosphere (JBI) [17], another distributed component architecture, which uses the publish/subscribe paradigm [14][18], provides many of the services necessary for a distributed component system. The JBI takes a slightly different approach than many other architectures. In the JBI, the flow of information from a data source such as a database to a client is made explicit. This information flow is dynamic and changes based on the request. Each component subscribes to and/or publishes information objects to the system. Using these publications and subscriptions, the JBI determines how to connect components together to form an information channel on which information flows. The JBI is responsible for setting up the information channels, managing them, and shutting them down. Therefore, developers that use the JBI approach can concentrate on the individual components of the system for publishing data, processing it in an information channel, and using the data in some user defined manner.

The idea of making the information flows explicit is not unique to the JBI. It was also studied by researchers at the IBM T.J. Watson Research Center in the context of distributed application services for Internet information portals [11]. The researchers proposed the use of data flow diagrams to represent the data processing that is required

starting from content providers to the Internet information portal on end devices. By capturing the content dissemination process, they were able to optimize the data flow and map the data flow onto physical application servers.

The JBI is a good architectural choice for redesigning mission critical systems. It makes the information flows in the system explicit. Many mission critical systems are information systems that channel information from a particular source to a particular sink through intermediaries, which manipulate the information. For example, a military system may send out information about all hostile bases in a large area. An intermediary component may filter this information for hostile bases in a smaller area. This information is then sent to the relevant clients. An advantage to making information channels explicit is that they can be customized and optimized as required.

Chapter 3. Background

In this chapter, background information on the Theater Battle Management Core Systems (TBMCS) is presented, along with relevant information about Internet technologies. TBMCS is an example of a mission critical system and was chosen as the focus of this study. Then multi-tiered systems, the Extensible Markup Language (XML) [8], the publish/subscribe network paradigm [14][18], and the Joint Battle Space Infosphere [17] approach are discussed in this chapter.

3.1 Theater Battle Management Core Systems

The Theater Battle Management Core System (TBMCS) is an integrated battle management system used to plan and execute an air campaign. It is a single integrated command and control (C2) system, providing a complete tool kit to plan and manage the overall air campaign.

TBMCS provides a wide range of support applications relating to the organization, personnel, procedures, and equipment; these applications are collectively known as the Theater Air Control System (TACS). The latter system links the various organizational levels of air command, control, and execution. TBMCS also facilitates intelligence operations, air battle planning, and execution functions for all theater air operations. The system provides functional connectivity horizontally to other services and allies, and vertically among air wings and other elements of the theater. TBMCS can accommodate the addition and deletion of information sources, operating units, available weapons, participating services, and participating allies.

Currently, TBMCS is a client/server UNIX system running on the Sun Solaris operating system. In this architecture, the client applications make procedure calls, using the common object request broker architecture (CORBA), to the database server and other applications to retrieve the information they need. The majority of these applications are written in C and C++. CORBA is used to provide location transparency. CORBA is a service that allows an application to make procedure calls on a remote application without knowing the location of that remote application. It also provides services for load balancing and fault tolerance. Figure 3-1 is an illustration of this system. There are several problems with this architecture. One problem is extensibility. The client programs communicate directly with the database servers. Therefore, a change to the database could force a change in the client applications. Portability is another problem. The same application cannot be run on different operating systems such as Windows NT and UNIX. A third problem arises when making updates to the software. Since most of the business logic resides in the client application, changes in the software need to be performed on each client application that has been deployed. All of these problems result in increased time and money in maintaining the system. A redesign of TBMCS is necessary to solve these problems.

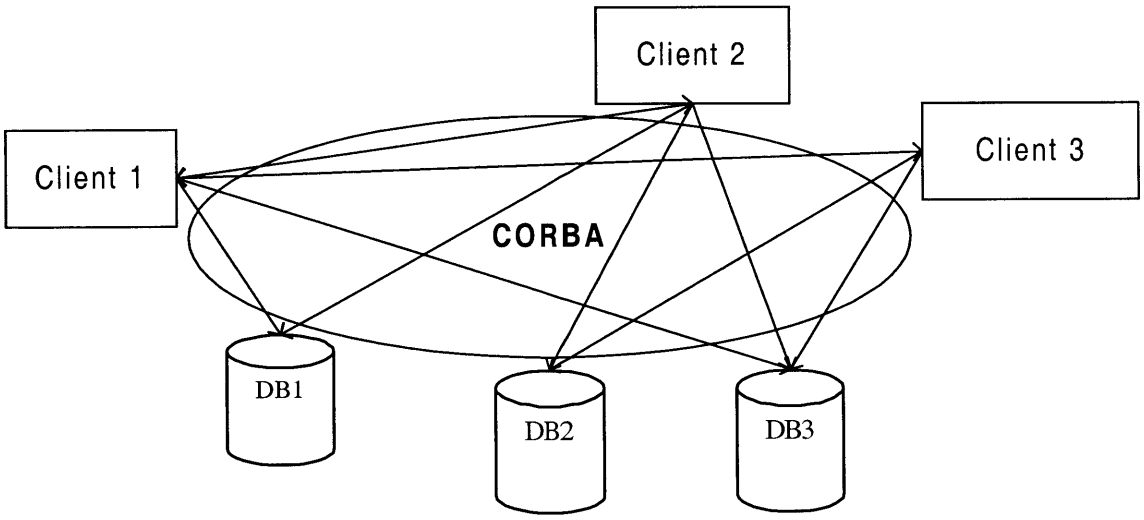


Figure 3-1. Current TBMCS architecture

3.2 Multi-tiered Systems

The emergence of technologies such as the Internet, Java, and XML has changed the way current systems are architected. Rather than the two-tier system where fat clients communicate directly with the backend systems like databases, a multi-tier architecture has emerged where thin clients prevail and an application server sits in the middle tier between the client and the backend. The middle tier may consist of several tiers, or just one tier, resulting in a system with several layers of abstraction. There are many advantages to the multi-tier approach. A major advantage is the decoupling of the client from the business logic. Most, if not all, of the major computation for an application resides on the application server. The clients therefore need only be concerned with displaying the results. Software updates can be performed in the middle tier(s) and all clients will receive them automatically the next time they access the application. Another major advantage of this architecture is the ability to reuse code. Many different clients can run the same application. By specifying the display format, the same application can

be run from a web browser and from a Java application. Finally, many of these multi-tier systems are being developed in Java. The use of Java increases portability, because most Java programs can be run on multiple operating systems without modification. A multi-tiered architecture solves many of the problems that exist in older client server architectures. A shift to a multi-tiered system could greatly reduce cost and increase maintainability and extensibility of the system. Figure 3-2 is an illustration of this so-called web architecture.

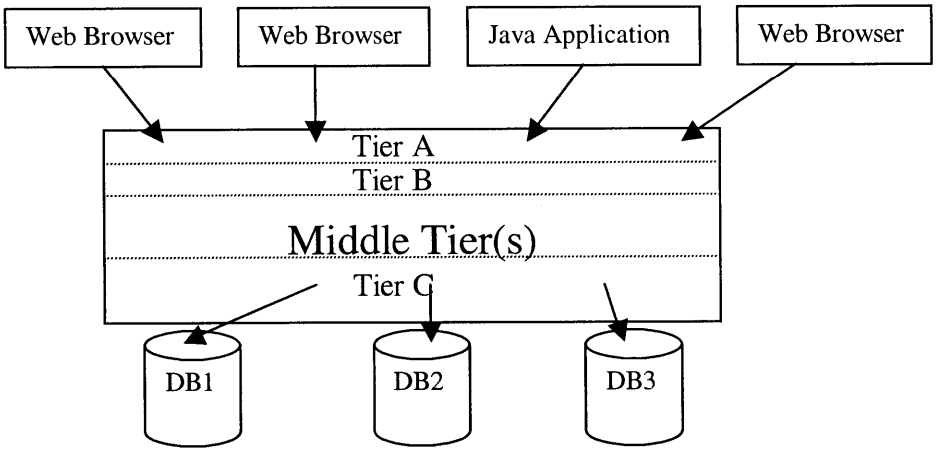


Figure 3-2. Web Architecture

3.3 Extensible Markup Languages

XML emerged out of a growing need to add structure to the wealth of unstructured information on the Web. HTML, the current language used to encode information on the Web, has several limitations. The most obvious limitation is its fixed tag set. It is impossible for Web developers to add custom tags in HTML [13]. Custom tags would be very useful when dealing with data pertaining to a specific domain. For example, a book publisher would prefer to use tags such as <title> and <author> rather than the generic <p> tag that is available in HTML. Furthermore, most web browsers render HTML

whether or not the file strictly adheres to the HTML syntax. For example, Netscape [25] displays the html snippets `<ol><li>apples<li>orange</ol>` even though it is the incorrect syntax. Therefore, for all practical purposes, HTML is only a loosely structured language.

The developers of XML looked at the Standard Generalized Markup Language (SGML) [26] as the basis for a more structured and extensible markup language. SGML is an international standard for the definition of device-independent, system-independent methods of representing texts in electronic form. It came out of IBM and became an International Standard Organization (ISO) standard in 1986. SGML is very powerful, offering three things that HTML lacks: extensibility, structure, and validation. However, SGML is also very complex and requires a significant amount of software overhead to process it. Due to its complexity and overhead, SGML was not used for representing hypertext in the Internet. HTML is actually a simplified application of SGML.

The XML 1.0 specification [8] was released by the World Wide Web Consortium (W3C) in February of 1998. XML was developed by a team of SGML experts whose goal was to create a new markup technology with the core benefits of SGML and the relative simplicity of HTML. XML is a metalanguage for designing markup languages. A markup language is a set of markup conventions used together for encoding texts. A markup language must specify what markup is allowed, what markup is required and how markup is to be distinguished from text [15]. A major advantage of XML is that one can easily create highly structured markup languages. Adding structure to large information

repositories such as the Web permits automation because applications can exchange this information and correctly parse and manipulate the data.

XML documents are made up of storage units called entities that contain either parsed data or unparsed data. Parsed data are made up of characters, some of which form the character data in the document, and some of which form the markup. The markup encodes a description of the document's storage layout and logical structure. XML provides a mechanism to impose constraints on the storage layout and logical structure. A software module called an XML processor is used to read an XML document and to provide access to its content and structure. An XML application employs the XML processor to gain access to the structure and content of XML documents [8]. A major component of the XML processor is the XML parser, which analyzes the markup and determines the structure of the document data. Most XML parsers can operate as either validating or non-validating parsers, depending on whether the XML document needs validation. The process of XML validation is discussed below.

All XML documents must conform to a schema, a model used to describe the structure of information within an XML document. XML schemas are similar to database schemas, which are used to describe the structure for storing data in a database management system. One can think of schemas as defining a vocabulary for the markup language being defined. There are two ways to specify the schema for an XML document: Document Type Definition (DTD) and XML Schema. Figure 3-3 shows an XML

Schema and a DTD for a phone book. Figure 3-4 shows the XML files that conform to the schemas in Figure 3-3.

PhoneBook.xml

```
<?xml version="1.0"?>
<Schema name=PhoneBook>
 xmlns="urn:schemas-microsoft-com:xml-data"
 xmlns:dt="urn:schemas-microsoft-com:datatypes>
 <ElementType name="firstname" content=textOnly"/>
 <ElementType name="lastname" content=textOnly"/>
 <ElementType name="number" content=textOnly"/>

 <ElementType name="entry" content="eltOnly">
  <description>
    This element represents a single entry in
    the phone book.
  </description>
  <element type="firstname" minOccurs="1"
maxOccurs="1"/>
  <element type="lastname" minOccurs="1" maxOccurs="1"/>
  <element type="number" minOccurs="1" maxOccurs="*"/>
 </ElementType>

 <ElementType name="phonebook" content="eltOnly">
  <description>
    This element type represents a phone book consisting
    of 1 or more entries.
  </decription>
  <element type="session" minOccurs="1" maxOccurse="*"/>
 </ElementType>
</Schema>
```

PhoneBook.dtd

```
<!ELEMENT phonebook (entry)+>

<!ELEMENT entry (firstname,
                 lastname,
                 number+)

<!ELEMENT firstname (#PCDATA)>

<!ELEMENT lastname (#PCDATA)>

<!ELEMENT number (#PCDATA)>
```

Figure 3-3. XML Schema (left) and DTD (right)

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE phonebook SYSTEM "PhoneBook.dtd">

<phonebook>
 <entry>
  <firstname>Bill</firstname>
  <lastname>Gates</lastname>
  <number>999-345-7654</number>
 </entry>

<entry>
  <firstname>Larry</firstname>
  <lastname>Ellison</lastname>
  <number>650-342-2786</number>
 </entry>
</phonebook>
```

```
<?xml version="1.0" standalone="no"?>

<phonebook xmlns="x-schema:PhoneBook.xml">
 <entry>
  <firstname>Bill</firstname>
  <lastname>Gates</lastname>
  <number>999-345-7654</number>
 </entry>

<entry>
  <firstname>Larry</firstname>
  <lastname>Ellison</lastname>
  <number>650-342-2786</number>
 </entry>
</phonebook>
```

Figure 3-4. XML file for XML Schema (left) and XML file for DTD (right)

As one can see from Figure 3-3, XML Schema is more verbose than DTD. However, it is also much more powerful and expressive. More information about DTDs and XML Schemas can be found at http://www.w3c.org.

One of the main goals of XML is to separate the content, which is the data, from the presentation, which is how the data is viewed. An XML document contains only the data, leaving it up to the application to handle displaying the data if necessary. Another XML technology, Extensible Stylesheet Language (XSL), was designed to handle the presentation of XML files. An XSL stylesheet specifies the presentation of a class of XML documents by describing how an instance of the class is transformed into an XML document that uses the formatting vocabulary. XSL consists of two parts, XSL Transformations (XSLT) and XSL Formatting Objects, which can be used independently. XSLT is the XSL transformation language that is used to transform XML documents from one vocabulary to another. XSL Formatting Objects is the XSL formatting language that is used to apply formatting styles to XML documents for display purposes. An example use of XSLT is to transform an XML document into an HTML file to be displayed on a web page. Another use of XSLT could be to transform an XML document into a tab-delimited flat file for use in another system that does not handle XML files.

A main feature of XML is the ability to access and manipulate an XML document programmatically. This can be done using the Document Object Model (DOM) API or the Simple API for XML (SAX). The Document Object Model is a platform- and language-independent programmatic interface that allows programs to dynamically access and update the content, structure, and style of XML documents. The DOM provides a standard set of objects for representing XML documents, a standard model of how these objects can be combined, and a standard interface for accessing and manipulating the objects. It is a combination of a tree model and an object model.

Vendors can support various DOM levels. Each level builds upon the previous level adding additional functionality. DOM levels 1 and 2 have been endorsed as a W3C Recommendation. DOM level 3 is currently a working draft. SAX is a standard interface for event-based XML parsing. It is at a lower level than DOM, offering more performance. It provides access to many aspects of XML processing that DOM hides. A major difference between SAX and DOM is that SAX is read only.

Earlier, the notion of XML validation was mentioned. XML documents can be classified as valid and/or well formed. All valid documents are well formed, but the reverse is not true. A document is valid if it is well formed and adheres to the schema referenced in the document. An XML document is well formed if it, among other things, has a single document element that contains all other elements, it has a start and end tag or empty tag, and elements are properly nested. The full definition of valid and well-formed documents can be found in the XML 1.0 Spec [8]. A validating processor checks the validity of an XML file, while a non-validating processor only checks whether the XML file is well formed. An application is not supposed to process a malformed XML file. This differs from HTML, where browsers often render syntactically incorrect HTML files.

3.4 Publish/Subscribe

Publish/subscribe is a middleware solution that is emerging as the best alternative for mission critical applications. In publish/subscribe [14][18] environments, there are producers and consumers. A producer publishes an information object of some type and

consumers subscribe to information objects of some type. When a producer publishes data of type T, all consumers that have subscribed to type T data are notified. The publish/subscribe mechanism solves the problem of clients constantly polling systems for changes in information. Constant polling is time consuming and increases network congestion. In the publish/subscribe framework, only information of interest to a client is delivered to the client and no polling by the client is necessary [16].

Two popular types of publish/subscribe systems are subject-based and content-based. Content-based publish/subscribe differs from subject-based in that content-based subscriptions can be made at a finer granularity. In subject-based systems, data are tagged with a single subject field and subscriptions are made with reference to the subject. In content-based systems, data have a metadata associated with them that subscriptions can be matched against. This allows the expression of more specific subscriptions. For example, in a subject based library publish/subscribe system, a subscription can be made for books about Latin dance. In a content-based system, a subscription for books about Latin dance published within the last year in the United States can be made.

3.5 Joint Battlespace Infosphere

The Joint Battlespace Infosphere (JBI) [17] is a combat information management system architecture that provides its users with the specific information required for their functional responsibilities during a crisis or conflict. It integrates data from a variety of sources, aggregates the information, and then distributes the information. This

information is presented to users in the appropriate form and level of detail. The data sources for the JBI include existing command and control (C2) systems such as TBMCS, as well as other sources like satellite and reconnaissance data. Users of the system can be a commander who wants a high-level view of the campaign, or a soldier who wants a detailed description of a nearby hostile base.

The JBI is a distributed component architecture based on the publish/subscribe paradigm. The system consists of a platform of protocols, processes, and common core functions that permit participating applications and organizations to share and exchange critical mission information quickly. It provides uniform rules for publishing new and updated objects into the JBI and alerts any JBI clients that have subscribed to such objects. These properties enable dynamic information flows among client programs of the JBI. In addition to publishing and subscribing, the JBI provides a query service that allows clients to search for information objects. The searching is done based on the metadata associated with the information objects. Figure 3-5 shows a diagram of the JBI.

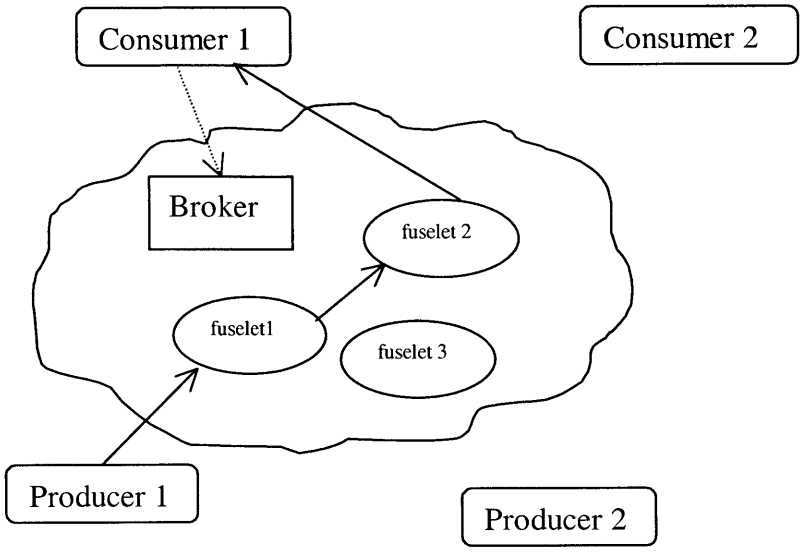


Figure 3-5. Joint Battlespace Infosphere Architecture

The main components of the JBI are the *broker* and the *fuselets*. Fuselets are JBI middleware applications. They create new knowledge derived from existing information objects and pass the new information object on to other fuselets or to the consumer. The broker is the component that receives the request from the user. It processes the request and determines how the fuselets should be arranged to get the information from the producer to the requesting client. This determination is based on the publications and subscriptions of the fuselets. The arrangement of fuselets is referred to as an information channel. Information channels can persist, or only exist for the length of a request. In Figure 3-5, Consumer 1 makes a request to the broker who determines that in order to fulfill the request, Producer 1, fuselet 1, and fuselet 2 need to coordinate. The Broker sets up the information flow and the data are then returned to the user.

Though the JBI was designed for use in the military, there is no reason the architecture could not be employed in a commercial environment. Here, online shopping is used as

an example. Assume a user has selected several items on a web page, which he or she wants to purchase. The user presses the submit button which should take the user to a page showing the list of products with the products' individual prices and the total price of all the products. The JBI platform can be used to implement such a web service. The list of products that was selected would be sent to the Broker. The Broker then determines from the request what fuselets are needed to fulfill the requests and connects them together. Figure 3-6 shows the information channel that results. Fuselets 1 and 2 are responsible for obtaining the data from the databases. Once they obtain the data, they format it and pass it on to fuselet 3. Fuselet 3 is responsible for aggregating the data from the first two fuselets and figuring out the price. Finally, fuselet 4 receives the data, which it then renders to HTML before sending it off to the client. Figure 3-7 shows the information object that each fuselet produces.

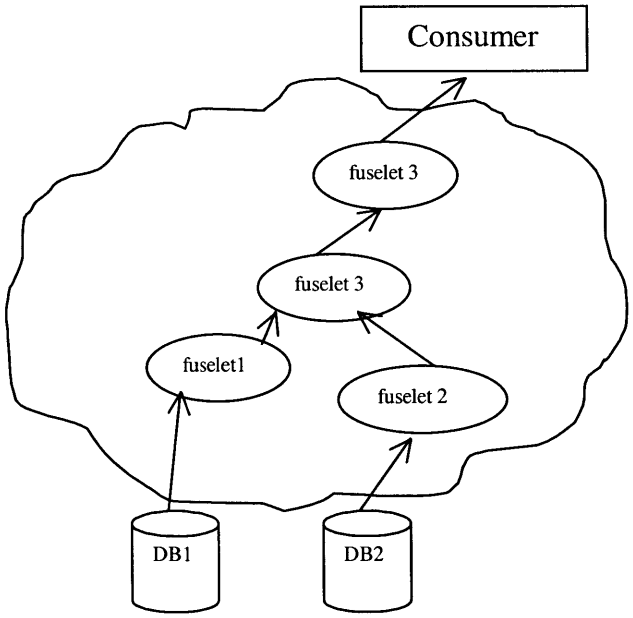


Figure 3-6. Online shopping information flow

Fuselet 1 (retrieves one product)

```
<products>
 <product>
  <name>Apple</name>
  <price>0.75</price>
  <currency>dollars</currency>
  <count>6</count>
 </product>
</products>
```

Fuselet 2 (retrieves one product)

```
<products>
  <product>
  <name>Orange</name>
  <price>1.25</price>
  <currency>dollars</currency>
  <count>4</count>
 </product>
</products>
```

Fuselet 3 (aggregates products)

```
<products>
 <product>
  <name>Apple</name>
  <cost>4.50</cost>
 </product>
 <product>
  <name>Orange</name>
  <cost>5.00</cost>
 </product>
 <total>9.50</total>
 <currency>dollars</currency>
</products>
```

Fuselet 4 (converts products to html)

```
<html>
<head><title>Checkout</head></title>
<body>
<table>
<tr><td>Product</td><td>Cost (dollars)</td></tr>
<tr><td>Apple</td><td>4.50</td></tr>
<tr><td>Orange</td><td>5.00</td></tr>
</table>

Your total is 9.50.
<form action=checkout.jsp>
<INPUT name=Submit type=submit value="Checkout">
</form>
</body>
</html>
```

Figure 3-7. XML files of information objects and final HTML file

Even in this simple example, the advantages of the JBI platform can be seen. The system is a multi-tiered system so all the benefits mentioned in section 3.2 are gained. In addition, the way the fuselets have been designed, there is a clear separation between obtaining the data, combining the data in intelligent ways, and formatting the data for a specific user. This makes the system extensible. If a new page is desired, the developer need only write a new aggregating fuselet and register it with the broker. No other changes to the existing fuselets need to be made. In addition, the new fuselet can use the results of other fuselets. For example, suppose one desired to make a different page that included the product prices along with some advertisements for related products. The new fuselet could use the results from fuselet 3 and then add the advertisement information. Another example is if a consumer wanted the same information to be

rendered on a WAP enabled mobile device. A new fuselet can be added that uses the result from fuselet 3 and converts the information to WML.

Another feature of the JBI platform is that it uses the publish/subscribe architecture for communication between producers and clients. This makes the system very extensible. New fuselets and clients can be added seamlessly by merely subscribing to the data it wants. No changes are needed to the existing modules.

Chapter 4. Related Work

There has been much work related to the design of distributed information systems. In this section, several research projects are presented that focus on efficient information dissemination and delivering custom information to clients.

4.1 An Infrastructure for Meta-Auctions

Cilia, Liebig, and Buchmann [7] worked on the design of online auctions, a popular trading mechanism where multiple buyers compete for scarce resources. In particular, their work concerned meta-auctions, a system that allows potential buyers to roam automatically and seamlessly across auction sites for auctions and items of interest. Their design is centered on a meta-auction broker that provides a unified view of different auction sites and services for browsing categories of items, searching for items, participating in auction, and tracking auctions. A main challenge they faced was providing communication protocols that allow efficient and transparent cooperation in a distributed heterogeneous environment. The design they came up with uses publish/subscribe based event notification and ontology-based meta-data management. The publish/subscribe paradigm was used because the request/reply nature of the Web is not adequate for timely notifying participants. By using publish/subscribe, bidders can find out when a higher bid is made and sellers can be notified when bids are made without polling the system. In addition, clients can be notified when a product of interest is placed on an auction. The ontology-based system is necessary because the system is heterogeneous. There are different users and auctions which all speak their own language. An ontology provides an agreement about a shared set of terms, or concepts,

of a given subject domain. The ontology serves as a common basis for the interpretation of data and notifications. Though this architecture is intended for online auctions, the underlying mechanism is applicable beyond auctions. The need for publish/subscribe and an ontology is present in any large-scale heterogeneous information dissemination system.

4.2 A Framework for Scalable Dissemination-Based Systems

Cilia, Liebig, and Buchmann used the work of Franklin and Zdonik [9] in their design. Franklin and Zdonik addressed the problem of building scalable dissemination-based systems. Data delivery mechanisms are distinguished along three main dimensions: push vs. pull, periodic vs. aperiodic, and unicast vs multicast. Existing data delivery mechanisms can be classified using the above characteristics. Request/response uses aperiodic pull over a unicast connection. Polling is a periodic or aperiodic pull over a unicast or multicast connection. Publish/subscribe is an aperiodic push often over a multicast connection, though it can also be unicast. The researchers propose the notion of a Dissemination-Based Information System (DBIS) that integrates a variety of data delivery mechanisms and information broker hierarchies. In their framework, you can swap in different data delivery mechanisms such as request/response and publish/subscribe. The DBIS consists of three types of nodes: (1) data sources, which provides the base data that is to be disseminated; (2) clients, which are net consumers of information; and (3) information brokers, which acquire information from other sources, add value to the information (e.g. some additional computation or organization structure), and then distribute the information to other

consumers. The researchers claim that by creating hierarchies of brokers, information delivery can be tailored to the needs of may different users. The information flow in the DBIS is very similar to that of the JBI. Information begins at a data source and then moves to the client through some information channel in which individual modules along the chain perform computation on the data before passing it on to the next module. The information brokers in the DBIS are analogous to the fuselets in the JBI.

4.3 A Self-Configurable Agent Architecture for Distributed Control

The coordination of components in distributed component architectures was studied by M. B. Blake at Georgetown University [5]. Distributed component architectures consist of individual components distributed across multiple machines. Each component performs an individual task to achieve an overall goal. Blake studied the use of rule-driven coordination (RDC) agents in such an environment. RDC agents act as brokers for the individual components. Each agent contains rules for the component it represents. The RDC agents are essentially a middleware layer between the distributed components. They encapsulate the interaction policy and the aspects of communication, data management, and policy execution. The decoupling of the communication, policy, and data management from the core logic of the component can be very useful. This design has proven effective in the distributed workflow domain. Many of the ideas in RDC agents are present in the JBI such as the notion of policy. However, the functionality is centralized. Decentralizing these services could result in a more robust and extensible system, and is therefore a possible area for further research.

4.4 An Automated Client-Driven Approach to Data Extraction using an Autonomous Decentralized Architecture

One of the goals of the system presented in this paper is to provide the user with custom information. Blake [6] had the same goal when he developed the MITRE Corporation-Center for Advanced Aviation System Development (CAASD) Repository System (CRS). CRS is an autonomous data extraction environment. At MITRE, there are several groups that analyze different problems, but the data to support the analysis are often the same. In addition, the format in which the groups want the data varies from specialized text files with delimited data to XML. Also, the groups look at multiple subsets of the data that may cross multiple data sources. The CRS system allows two main functions for users. They can design and save a personalized query form. This is useful for people that need to execute repetitious personalized queries. The other function is accessing a personalized or standard query form and executing a query on the data repository. The design of this system has four main modules: the Client Interface Module (CIM), the Interface Specification Module (ISM), the Presentation and Query Module (PQM), and the Database Extraction Module (DEM). These modules are placed into three layers: (1) the Interface Layer, which contains the CIM, (2) the Presentation Layer, which contains the ISM and PQM, and (3) the Data Storage Layer, which contains the DEM. The Interface Layer is the layer by which users connect to the system. Web browsers are a type of Client Interface Module. The Presentation Layer includes the software services that provide a graphical user interface. The ISM gives customers the ability to customize their user interface to meet their specific needs. This accounts for the diverse data needs. The PQM lets customers choose a standard or specialized interface in order to request data. In the future, it will also allow the specification of business and

domain logic. The Data Storage Layer contains the functionality to maintain and extract data from some data repository. It consists of software services for extracting data from a relational database management system (RDBMS). This architecture is very flexible and permits a client driven approach to data extraction. A similar interface to the system described in this paper would be very useful in terms of getting the user the data they want, the way they want it. There is ongoing work in the JBI team to add such functionality.

4.5 Information Flow Based Event Distribution Middleware

Event distribution middleware supports the integration of distributed applications by accepting events from information producers and disseminating applicable events to interested consumers. The integration is achieved using the publish/subscribe paradigm we described above. Banavar, Kaplan et al. [2] presents a flexible new model, the Information Flow Graph (IFG), for specifying the flow of information in such a system. An IFG contains two components: information spaces and dataflows. Information spaces are either event histories, which are lists of events, or states, which capture selected information about event streams. For example, in a stock-tracking example, the New York Stock Exchange would be an event history information space and the maximum price of the day for Microsoft stock would be a state. Dataflows are directed arcs that connect the information spaces. The arcs determine how the contents of the information spaces change as events enter the system. Sources have only out-arcs and sinks only have in-arcs. There are four types of arcs: select, transform, collapse, and expand. The select arc connects two event histories having the same schema. Traversing a select arc

carries a subset of the source data to the sink. The transform arc connects two event histories, which may have different event schemas. It maps events expressed in one schema to the same event expressed in another schema. The collapse arc collapses a sequence of events into a state. The expand arc reverses the collapse arc linking a state to an information space. The merging of events (fan-in) and the dissemination of events (fan-out) are implicit operations in the graph. Using a flow graph rewriting optimization, stateless IFGs can be converted into a form that can exploit efficient multicast technology developed for content-based publish/subscribe.

Many ideas in the IFG are present in the design of the car service system. Producers and data fuselets are the information spaces. The transform arcs are represented by the formatting fuselets. The select arcs are the filtering fuselets. The IFG rewriting works because the transforms and selects can be expressed via rules. Expressing the complex interactions between fuselets as rules is difficult. However, if the rules for such a system could by constructed, the rewriting of the IFG may give insight into a more efficient implementation.

The design of this system did not focus on performance, though it was kept in mind. An example of that is the decision to combine several formatting/filtering fuselets into one fuselet to improve speed. Banavar, Chandra et al [1] studied efficient protocols to improve performance in a content-based publish-subscribe system. In a subject-based publish-subscribe system, matching publications with subscriptions and multicasting events are amenable to straightforward solutions. Matching can be done with a simple

table lookup of the subjects. The multicasting problem can be solved by defining a multicast group per subject and sending each event to the appropriate group. Content-based systems do not have such straightforward solutions to the matching and multicasting problems. The researchers present an efficient solution to the multicasting problem, called link matching. Each component in the system is considered a broker, which are connected in a network. Each broker acts as a router, which can forward events to other brokers. In the link matching algorithm, each broker performs just enough of the matching work to determine which neighboring brokers should receive the event, and then forwards the event along links to these neighbors. This approach is compared to a flooding algorithm where events are sent to every broker and only those interested in the event process it. The link matching algorithm outperformed flooding on tests using computer simulation and a prototype. The researchers claim that the link matching algorithm is better than a point-to-point algorithm in which events are delivered directly to each interested broker. The reasoning behind this is that in the point-to-point algorithm, an event may be sent over the same link twice, which is wasteful.

The link matching approach differs from the JBI approach in several ways. In the JBI, it is assumed that each component can establish a direct connection with each other component. In other words, the network of components is fully connected. Therefore, there is no need for each component to act as a router that passes on events. Events can simply be delivered in a point-to-point protocol. In [1], it is assumed the network is not fully connected. Therefore, brokers must act as routers to ensure that events can be delivered to all brokers. Since the JBI architecture allows direct connections between

components, the link matching algorithm would not be useful. However, if during implementation, it is discovered that allowing a direct link between components is too costly, a link matching type algorithm may be very useful in improving the performance of the system.

Chapter 5. Design

The details of the Theater Battle Management Core System are proprietary. Therefore, it is not possible to discuss the specifics of the system in this paper. For this reason, a cab service analogy was devised. The car service (CS) system described below can be mapped back to TBMCS.

5.1 Design Criteria

Liskov [12] states that the purpose of design is to define a program structure consisting of a number of modules that can be implemented independently and that, when implemented, will together satisfy the requirements specification and meet the performance constraints. She claims that a good program structure is reasonably simple, avoids duplication of effort, and supports both initial program development and maintenance and modification. This definition of good design was kept in mind throughout the design process.

The basic requirement for the CS system is to determine the daily schedule for car drivers, to send them their schedule for the day, and to modify it as changes occur. This schedule should include information about the pickup location, directions on how to get from the pickup location to the destination location, images, and maps. The drivers should also be sent general information such as weather reports, news reports, and sports scores. The system should be able to support both pulling and pushing information. An example of pulling information is a car driver requesting his or her schedule for the day.

An example of pushing information is a car driver being sent an updated weather report when one becomes available.

Though performance is of great importance in such a system, more emphasis was placed on extensibility. Any large system is going to evolve over time; and an extensible system facilitates this evolution by supporting both program maintenance and modification. In addition, performance is largely an artifact of the underlying architecture on which the system is built, in this case the JBI. The speed of this system is dependent on how quickly information channels can be set up, the speed of the individual components, and the network that connects the distributed components.

In terms of extensibility, this design aims to allow the easy addition of new information channels without altering existing channels. More specifically, new clients should be able to tap into existing information channels with little work, modules should be reused as much as possible to reduce duplicating work, and new channels that use part of another channel should be easily constructed to promote reusability. The easy addition of new channels allows the system to expand as new requirements arise. The design also tries to cater to a large variety of users. It accommodates users that require the same data in a different format and users that require the same raw data, but would like it manipulated differently.

5.2 Car Service (CS) System

There are two car service companies, the Boston Car Service and the Cambridge Car Service. All of the cars in the companies have computers on board and are outfitted with Global Positioning Satellite (GPS) systems that indicate where the cars are located. The car computers of both companies are equipped with browsing applications, but the applications in Boston Car Service take a different format than the applications in Cambridge Car Service. The car service system is a publish/subscribe system that consists of several data sources and services that the drivers use to effectively respond to customer requests. The main goal of the system is to get the car drivers their schedule for the day, to monitor the progress of the cars and update the schedules accordingly, and to respond to new requests for service. The following is a listing of the individual modules present in the system. The components are classified into one of three categories: Data Producer, Service, or Client.

5.2.1 Data Producers

Road Construction

The road construction producer contains information about roadwork that is scheduled to occur. This data are used by the route finder to avoid routes that go through construction areas.

Geographic/Mapping Data (GIS)

The GIS producer contains information about roads such as location, how roads interconnect, one/two-way, highway/local road, number of lanes, etc. The producer also

contains maps such as street maps and elevation maps. This module is used by the route finder to determine routes and is also a source of maps for the map overlay service.

CNN and MSNBC Traffic Reports

These producers contain current and predicted information about traffic on roads. Information can be received by requesting data for a specific road or providing a general area. These modules are used by the traffic reports consolidator.

Weather Reports and Maps

This module contains current weather reports and weather forecasts. The module also contains various weather maps. It is used by the car schedule fuselet to provide weather data to drivers. It is also used by the map overlay service.

Car Information

This module contains information about individual cars such as size, model, etc. It also contains the cars status (in use or free) and its estimated time of arrival if the car is currently servicing a request. It is used by the car planner service to assign cars to pickups.

Location Information

This producer contains information about locations such as capacity, hours of operation, population at given points in time, etc. A location can be anything from the Marriott Hotel to a gas station. Location information is used by the location folder service.

Images And Video

The images and video producer contains images and videos of various locations. The images include both exterior and interior shots. Cars equipped with cameras can send images and videos to this module. This module is used by the location folder service.

Scheduled Pickups

The scheduled pickups module contains the pickups that need to be made. These are requests that were made in advance. A pickup consists of a time, pickup location, destination location, amount of luggage, and the number of passengers. This producer is used by the car planner service to assign cars to pickups.

Sports Scores

The sports scores producer contains up to the minute sports scores. It is used by the car schedule to provide drivers with updated scores.

5.2.2 Services

Map Overlay

This module takes in maps as its input and overlays them. For example, a street map can be overlaid with a weather and elevation map to determine if an uphill road has received a large amount of snow accumulation. If it has, then a car driver should avoid that road. The map overlay service is used by the location folder service.

Traffic Report Consolidator

The inputs to this service are various traffic report producers. The traffic report consolidator merges the information to form a consolidated traffic report. When the information from the traffic report producers conflict, the conflict is resolved via some heuristic. One heuristic could be to take the report that has the later timestamp. It is used by the route finder to help estimate the time to traverse a route.

Population over time

This module uses the population data of buildings from the location information service and gives the estimated population at a given time and the estimated changes over time. This module is used by the hot spots discovery fuselet. A building that is about to experience a sharp decrease in capacity is likely to have customers outside of it.

Hot Spots Discovery

This module takes as its inputs information from car drivers, the traffic report consolidator, and the population over time service. It uses these data to come up with a list of places a driver can drive by that have a high likelihood of having customers. The locations in the list are called hot spots. The traffic reports are used to filter hot spots that happen to be in high traffic areas. Car driver can send this module hot spots that were discovered while driving around.

Car Allocator

This module is used to respond to an ad hoc request. Based on car schedules and their current location, it determines which car should be used to service a new request. It uses the hot spots module to avoid sending a car away from a hot spot. The route finder is used to determine how long it would take for a car to get to the pickup location.

Route Finder

The route finder service receives information from the traffic reports and the mapping data and comes up with the best route between two locations. "Best" is measured in terms of the least travel time based on the normal flow along the roads and the traffic reports. This service is used by several other services, such as the location folder and the car planner.

Location Folder

The location folder aggregates information about a given location. It uses several other modules for retrieving the information. For example, a location folder for the Wang Center could include Wang Center information, Wang Center images, the fastest route from the Wang Center to the Eliot Hotel, and a map of the route overlaid with a weather map.

Car Planner

The car planner is used to figure out what each car should do for the day. In coming up with the schedule for the cars, the goal is to maximize profits. This includes maximizing

pickups, minimizing idle time, minimizing trip time, etc. To perform this function, it first gets the scheduled pickups. It then uses the route finder, car information, road construction, weather reports, and location information to find the optimal scheduling.

Car Schedule

This module is responsible for delivering the car schedules to each car. It receives the schedule from the Car Planner. It then retrieves a location folder for each scheduled pickup location. This information along with some general information is packaged together and is sent to the car computers. General information includes weather forecasts and sports scores.

5.2.3 Clients

Monitoring

This application monitors the entire system. Car drivers report their progress and the result of their trips to the Monitoring module. The module is like an administrator of the system. It can make changes such as reassigning a pickup to a different car.

Cars

Car drivers receive their daily schedule from the car schedule fuselet and report back their progress and the results of the trips to the monitoring client. Drivers can also inform the hot spots discovery service of new hot spots that were found.

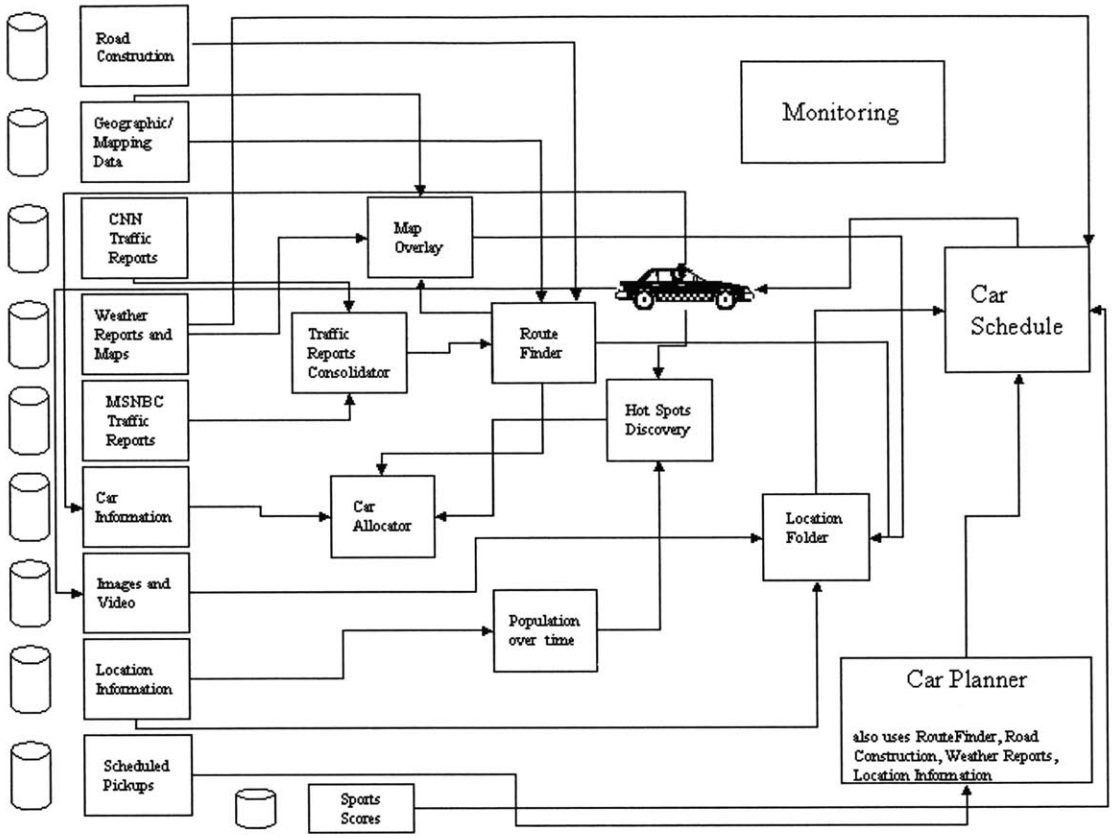


Figure 5-1. Car Service System Information flow

Figure 5-1 shows the information flows that exist in the system. The Data Producers have a database associated with them to illustrate that they are serving up raw data. The Services take data and performs some operation to produce new data. The Clients are the car and monitoring modules. The designation of the components directly maps to JBI components. The data producers correspond to JBI producers. The services are fuselets and the clients are consumers. In Figure 5-1, there is only one copy of each module. This may not be the case when the system is implemented. One possibility is that there will be multiple road construction producers to increase availability and distribute the

load. Another possibility is that each car company will have its own car information module.

5.3 Explanation

The data required to determine each cars schedule are located in the producers. These data needs to be combined and manipulated to determine an optimal scheduling of cars. The goal of the car service (CS) system design is to transform the data in the producers to a schedule, while adhering to the tenets of good design such as modularity and extensibility. This is necessary to allow relatively simple system updates. For example, in the CS system, there is a route finder that determines the best route between two locations. If a more efficient and reliable route finding algorithm is devised, a fuselet that encapsulates this new technology can be used in place of the old route finder without altering the rest of the system. If the Route Finder had not been a separate module and had been placed inside the GIS producer, then using the new technology in the car service system would be more difficult. Also, bundling the route finding with the GIS producer would make it difficult to allow traffic reports to be considered when determining the best route.

The central module in the CS system is the car schedule fuselet. The car schedule fuselet is responsible for sending the car drivers their schedule of pickups for the day along with auxiliary information to help them on their trips. To achieve this goal, the fuselet first gets the car plans from the car planner fuselet, which contains the mappings between each car and their pickups. The car schedule fuselet then takes each pickup location and

obtains a location folder for it. The location folder contains information about the location such as the address, pictures, maps, and directions. The location folder along with general information is sent to the car assigned to make a pickup at that location. General information includes weather reports and sports scores among other things. The car uses all this information to efficiently complete its assignments. The connections between the modules are made by the broker based on the subscriptions and publications of each module.

Though the car schedule is the main module that the cars receive information from, drivers can go directly to the information source and receive information if the producer or fuselet allows it. In the CS system, information sources such as producers and fuselets can have a web interface through which a user can make requests. It is also possible for a client application to be installed in the car that hooks into the information source through the programmatic interface. The client application would provide a user interface to the data and services of the producer or fuselet. This way, drivers can get more information from the system that is not provided by the car schedule. For example, a driver can go to the map overlay fuselet to obtain more maps than were not provided in the schedule.

5.4 System Dynamics

In this section, the dynamics of the system are illustrated. The information channels described below only represent one possible configuration of the system. Many other configurations are also possible. It may be useful to refer back to Figure 5-1 while reading the description.

Assume the producers have all been loaded with data. At some point, the system is run to compute the schedules for the cars in the system. The flow of information begins at the producers. The CNN and MSNBC traffic report modules send the traffic reports consolidator their respective traffic report predictions. When the traffic reports consolidator receives these reports, it consolidates them, resolving any conflicts that may arise. This information is then sent to the route finder.

While the traffic reports are being generated, the location information and images and video producers are sending information to the location folder fuselet. The GIS producer and road construction are also sending data to the route finder. Once the route finder receives data from all of its inputs, it begins computing the necessary routes. When the routes are computed, they are sent to the location folder and the map overlay. The map overlay fuselet overlays maps of the routes with a weather map. The maps are then sent to the location folder. The location folder fuselet sorts through all of this information, making location folders for each route that it receives that includes the direction, the location information of the source, images of the source and destination, and a map of the route overlaid with a weather map.

While all of the above is happening, the car planner receives the scheduled pickups for the day and car information. It uses this, along with other data from the system, to schedule the cars.

Both the location folder and the car planner send their results to the car schedule fuselet. The car schedule fuselet also receives data from the sports scores and weather reports module. It uses these data to send each car its schedule for the day.

If during the day a new request is made, the car allocator is run. The car allocator retrieves data from car information, hot spots discovery, and the route finder. This information is used to determine which car should service the request. The car schedule is then notified of this change and the car is sent an updated schedule.

Throughout the day, a driver that does not have anything scheduled can request a list of hot spots. This list is retrieved from the hot spots discovery module that uses the population over time fuselet to determine the hot spots list.

5.5 Push versus Pull

An issue that has not yet been addressed is whether information is pushed or pulled along the flow. The appropriate mechanism depends on the needs of the client and the frequency of the information changes. Take the route finder fuselet for example. It uses the GIS data, along with information about planned road construction and predicted traffic reports. Assume that after the schedules are given out, the traffic report fuselet changes its prediction. If speed were of the utmost important to a driver, then the driver would want that new traffic information pushed to the route finder, which would use it to determine a new route. This new route would then be sent to the driver in an updated schedule. However, if there is a driver who is not pressed for time, then there is no need

to provide him with an updated schedule. Providing unnecessary updates will just slow down the system. There could also be a case in the middle in which a driver does not care about a new route that saves ten minutes, but does care about saving more than ten minutes. This driver would want the traffic report fuselet to forward the new report to the route finder, but for the route finder to forward a new route only if it improves the old route by at least ten minutes.

Another example involves the image producer. Assume a location folder has been given out for the Wang Center that contains images of the theater. If new images of the Wang Center were being added to the producer constantly, then it would not be a good idea to update the folder when each new image is added. This would increase network congestion slowing down the system. If however, images were added every hour or so, then it may be worthwhile to update the folder with the new image. Once again, there is a middle ground in which image updates are sent periodically and the length of the period is determined by the client. Similar scenarios can be devised for each other information producer in the system.

In the CS system, each module has the capability of pushing its data or responding to requests (pulling). The pushing of data is flow specific. This means that one can specify that one car going to the Wang Center receives constant image updates and another car only receives updates every hour. The parameter(s) with which the push can be specified varies for each fuselet. For example, a route finder fuselet would have a parameter that specifies the amount of time saved in the new route.

5.6 Module Types

The modules have so far been divided along three dimensions based on their roles in the information flow: data producer, service, and client. They can also be divided along the function that they perform. Some modules are computationally intensive such as map overlay and route finder. Others are related to knowledge discovery such as hot spots discovery and population over time. The location folder deals with aggregating data and the cab schedule module deals with presenting the data. Aside from these functions, there are three important jobs that constantly need to be performed in the CS system: formatting, filtering, and data consolidation.

5.6.1 Formatting and Filtering

One of the goals of this system is to cater to a large variety of clients. In particular, the system should support clients that want the same data in different formats and clients that want different subsets of the same data. The former is referred to as formatting and the latter as filtering. The goal is to support all of these different users using as little work and as much module reuse as possible.

When data comes out of a producer or a fuselet, it is given in a specific format. Often this format is not the one desired by the user. There are three options available to transform the data from one format into that required by the client. One option is to have the producers and fuselets deliver the data in every format desired. This takes the burden completely off of the client, which must now merely request the data in the desired

format. However, there are a couple of problems with this approach. One is that forcing the producer of the information to format the data for every user adds complexity to the producer or fuselet, which may already be complex due to the functionality it provides. A second issue arises when trying to handle the addition of a new client to the system that needs a new format. If a client comes along that requires a format not currently produced, the client may have to wait a long time until the producer or fuselet adds support for that format.

A second option is to force the client to do all format conversions. This removes the burden completely from the producer or fuselet producing the data and places it on the client of the data. If a client requires some format, it must write a wrapper that converts the format produced into the format desired. The problem with this solution is that it may result in work being duplicated. For example, if 50 clients all use the same data in the same format, then it is a waste for each of them to reformat the data coming from the producer. It would be better if this reformatting was done once and passed to each client.

The first two solutions push the formatting responsibility to the ends of the information channel. A third solution places the burden in the middle of the channel onto a special fuselet. A formatting fuselet is used that has the sole purpose of converting data from one format to another. This solves the problem with the second solution, because reformatting is done once and is sent to all clients. It also solves the problems with the first solution, because a fuselet can be added in less time than it would take for the producer to support the new format. There are several ways to implement this strategy.

One is to have a fuselet for each data/format pair. This is very flexible, but may result in many fuselets. A large number of fuselets could negatively impact the performance of the system. Another option is to have one formatting fuselet per producer or fuselet that handles all of the conversions. This option is good because the formatting fuselet can reuse the information about the original format for each new format being produced. This solution however shares the same problem as the first solution, where supporting new formats may take a while. A third option is to have a fuselet per format. This would however result in a monolithic fuselet that has knowledge of every type of data being produced in the system.

As can be seen from the preceding discussion, each solution has tradeoffs. If the number of fuselets is not an issue, then having a fuselet for each data/format pair may be the best option. If reformatting is trivial, then adding support for all the formats inside the producer may be best. Given all of these factors, the following scheme was devised to handle formatting data. The scheme is a combination of the three solutions above.

When producers and fuselets are designed, they support a set number of popular formats. If one module comes along that requires a different format, it must do the conversion itself. The reasoning behind this is that if the module is the only application requiring the format, then it is not worth adding a fuselet or altering the existing modules. If however, several modules come along that require a format that is not currently produced, then a fuselet is added to the system that converts data from the old format to the new one. If we forced each of the new modules to reformat the data, work would be duplicated

unnecessarily. It is possible that over time there will be too many formatting fuselets causing the system to slow down. If this is the case, then fuselets that format the same data can be merged into a larger fuselet that supports multiple formats. Also, the producer may decide that it is worth the effort to support additional formats directly. Then the formatting fuselets for those formats can be removed. This scheme promotes extensibility when it is deemed worthy, and allows for performance optimizations if necessary.

All of the discussion so far has dealt with formatting, but the same argument holds for filtering. The scheme just described is also used to handle filtering data.

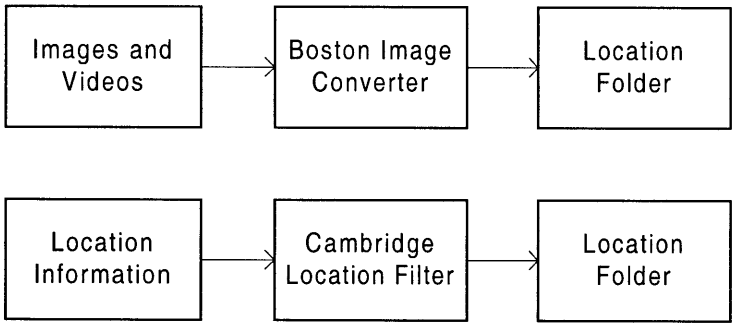


Figure 5-2. Filtering and formatting fuselets

Figure 5-2 shows how the filtering and formatting fuselets fit into the CS system. There is a Boston Image Converter fuselet and a Cambridge Location Filter fuselet. The Boston cab applications use a different format for their images than what is produced by the Image producer. Therefore all images that are sent to Boston cars must first pass through the Boston Image Converter. The use of a fuselet in this situation is consistent with the scheme described above. There are many clients (cars); therefore a separate formatting

fuselet is necessary. The Cambridge car company also has specific needs. It is not interested in all the data that the Location Information producer provides about locations. Therefore, there is a Location Filter fuselet to remove the unwanted information. Once again, the separate fuselet is warranted because there are many clients that desire the filtering.

5.6.2 Data Consolidation

Frequently, there are multiple sources of similar data. These sources contain the same information, so a fuselet or consumer can go to either producer to receive the data they want. In Figure 5-1, there are two sources of traffic reports, CNN and MSNBC. Unlike the previous case, these producers do not necessarily have the same data. The area coverage of CNN and MSNBC may not be the same. Therefore in some cases one can only go to one of the sources to get the desired traffic report. If instead the request is for an area in which the sources overlap, it may be necessary to perform conflict resolution if the two sources give different reports. The conflict resolution algorithm can be arbitrarily simple or complex. Two example conflict resolution strategies are using prior knowledge of the reliability of the sources and choosing the most recent report by checking the timestamps.

A client of the traffic report data should not have to be concerned with determining which producer to go to and resolving conflicts between producers. Ideally, a client would make a request to one module for the traffic report in an area, and the correct report is returned. To achieve this, a fuselet is needed as an intermediary to the traffic reports.

The function that this intermediate fuselet performs is referred to as data consolidation. It receives data from the multiple sources and uses them to respond to all traffic report requests. Data consolidation is needed to simplify clients when multiple sources of the same data are present. Presenting a layer of abstraction between the clients and the data sources also allows for additional sources of the data to be added without knowledge to the client.

5.7 Data presentation

A main goal of the CS system is to support many clients with varying needs. The formatting and filtering mechanism was designed to achieve this goal. Once the data have been acquired, they need to be presented to the client. Since clients would like to view data differently, a separation was made between the data and the presentation of the data. Each module provides an XML format of the data that does not include any presentation information. Display information is then added by a presentation module.

The main module that deals with presentation in the CS system is the car schedule fuselet. It takes the data from the location folder, the car planner, weather, and sports scores and outputs a file that contains presentation information. This file could be anything such as HTML, WML, or even ASCII art. The file format that a client receives is determined based on the request or from prior knowledge of the client's needs. Other modules can provide a presentation format of their data, but this feature is completely optional.

In the design of the data presentation, the driver experience should be kept in mind. The data should be presented in a way that is intuitive and allows the driver to access the information quickly. The presentation of the data described in this paper is just one example of how the data can be presented. More elaborate presentation schemes are encouraged. For example, a car schedule can be read to a driver through a voice-enhanced application.

Figure 5-3 shows a set of linked pages for displaying the information. The first page is the schedule homepage. It contains links to each scheduled pickup, as well as general information links. General information links point to pages that have information that is not location specific, such as sports scores. The pickup link takes you to the location folder information. This page contains links to each location folder item such as location information, directions, and images. This data organization was chosen so that a driver can easily scan its scheduled pickups for the day, and drill down to find more information about the pickup.

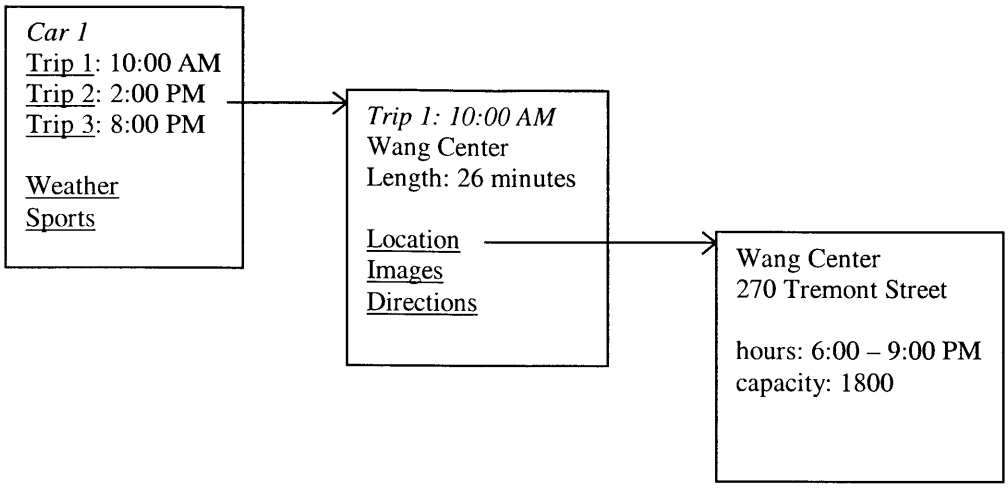


Figure 5-3. Sample car schedule pages

5.8 Setting up Information Flows

So far the discussion has been about the information channels themselves. There has been little talk of the how the information channels are set up. This is the responsibility of the Joint Battlespace Infosphere (JBI). The JBI provides a profiling language in which publications and subscriptions are made. Clients subscribe to the information they require, producers publish the information they produce, and fuselets subscribe to the data they require and publish the data they produce. The broker uses these publications and subscriptions to set up and manage the information channels. The profiling language allows the specification of a variety of information channels. Take the car service system for example. The car schedule fuselet subscribes to daily car plan data creating a pull channel. The route finder subscribes to changes in traffic reports creating a push channel. The car schedule fuselet subscribes to location folders each day. During the day, changes in the folder are pushed to car schedule. When the day is over, the channel is removed. This is an example of a push channel that only exists for a specified amount of time. Many other types of information channels are also possible using the profiling language

5.9 Mapping the CS Design to TBMCS

As stated at the beginning of the chapter, the car service system can be mapped back to the TBMCS system. A detailed mapping of the CS system was made for the TBMCS project team. In some cases, translating CS modules to TBMCS modules was as simple as changing the name. In other cases, the TBMCS modules were represented as multiple modules in the CS system. There are some CS system modules that have no TBMCS counterpoint. The existence of these components in the CS system suggested components that could be of use in TBMCS.

Chapter 6. Prototype

In this chapter, the prototype of the design described in Chapter 4 is presented. A scenario was invented to demonstrate the main features of the system. This scenario motivated the choice of the components and the information channels that were prototyped. The implementation of each of the prototyped modules and channels are explained. Two third-party software applications were used in the prototype and both of them are described along with the reasons for choosing them.

6.1 Scenario

The goal of the car service (CS) system prototype is to demonstrate the main features of the design. A scenario was therefore devised that illustrates the main functionality of the system. The scenario was used to choose the modules and information channels to prototype.

There are two car service companies: the Boston Car Service and the Cambridge Car Service. Each car service has one limousine and one sedan. The producers that exist in the system are the GIS producer, MSNBC traffic reports, CNN traffic reports, location information, car information, scheduled pickups, sports scores, and images and video. These producers are used by the fuselets route finder, traffic report consolidator, location folder, car planner, and car schedule. The car schedule delivers information to the consumer the car computers. The computers in the Boston cars take a different image format than the computers in the Cambridge cars. The Cambridge computers read jpeg images, but the Boston computers only read gif images. The images producer produces

only jpeg images. The two companies also differ in that the Cambridge drivers are not interested in all the location information that the location information producer delivers. Cambridge car drivers do not want to know about the function of a location. Figure 6-1 is an illustration of this system.

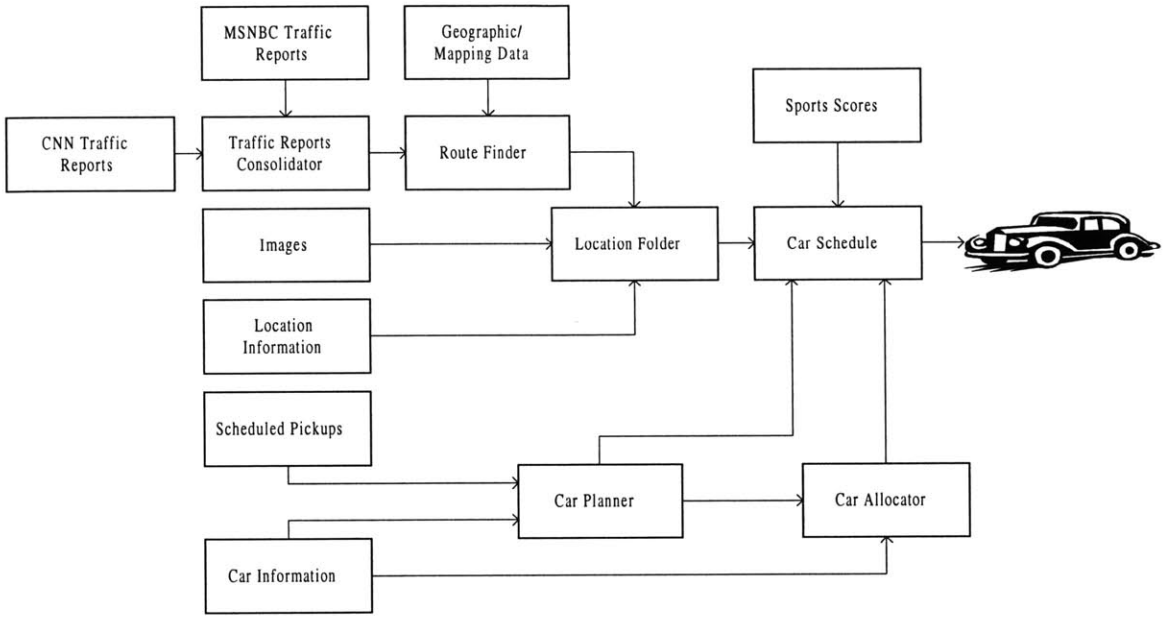


Figure 6-1.

From	To	Time
Brian's House	Museum of Fine Arts	12:00 PM
Fenway Park	John Harvards	5:00 PM
The Eliot Hotel	Legal Seafood	6:00 PM
Legal Seafood	The Eliot Hotel	8:00 PM
Tang Hall (MIT)	Lumiere Restaurant	5:00 PM
Lumiere Restaurant	Wang Center	7:00 PM
Wang Center	Ryles	10:00 PM

Table 6-1. Scheduled pickups for the day

Table 6-1 shows the pickup schedule for the scenario. At the beginning of the day, the car planner is run and the car schedules are determined. Each driver's schedule is sent to

his or her car computer. Every so often new images and sports scores are made available and the data is sent to the appropriate cars. At some point during the day, the traffic report consolidator changes its prediction about traffic. This causes the route finder to reevaluate the optimal route it has computed. If the optimal route changes, the new route is sent to the appropriate car. Later a request comes in for a pickup, and the car allocator is run. It determines which car should service the new request and updates that car's schedule.

6.2 Third party applications

Two third-party systems were used for this prototype: the Information Personalization Agent Manager (iPAM) and a GIS data/route finding application.

6.2.1 Information Personalization Agent Manager (iPAM)

Ideally, the prototype of this system would use the Joint Battlespace Infosphere (JBI) as the development environment, because the system is designed for the JBI. However, the JBI is still in the development phase. Therefore, the Information Personalization Agent Manager (iPAM) [20] was chosen as the development environment. IPAM was developed at the MITRE Corporation. It is a software framework that facilitates building information channels. IPAM provides an infrastructure for connecting modules together to form an information flow. It provides several agents that can be used. The framework also provides an API so that developers can plug their own agents into the iPAM system. The system is written entirely in Java. IPAM was chosen for its availability and its ability to easily connect agents together through a graphical user interface.

IPAM consists of 5 main components: input agents, output agents, handlers, daemons, and the executive. Input agents are the entry point of information in the system. They are responsible for obtaining the information and converting it into the appropriate system format. Inputs can be synchronous or asynchronous. Synchronous input agents send data when a request is made, while asynchronous input agents push data along the information flow as well as respond to requests. Output agents are the exit point of the system. They receive data from the system and are responsible for formatting and disseminating the information to clients of the system. Handlers perform the internal processing of the information. They take a single or multiple inputs, process the data, and output the result. Handlers can have a schedule associated with them so that they run at a specified time. By connecting handlers to one another, an arbitrarily long information channel can be formed. Daemons are agents that have a schedule associated with them. They differ from handlers in that they do not have inputs or outputs. A daemon can be used to check and clean up the information channels at regular intervals. Finally, the executive is the heart of iPAM. It manages the system and provides the services that the agents use. All of the 5 components, with the exception of the executive, have properties that can be set from the iPAM user interface.

IPAM provides several administrative tools allowing administrators to build information collection and dissemination services through a graphical user interface. Once the administrator decides what agents are needed, he can add them to the system and set their properties from a user interface called Bart. Connections between agents can be made by

clicking on the window. Figure 6-2 shows a picture of this interface. IPAM provides several other tools such as the Watcher that shows which agents are active as the system is running.

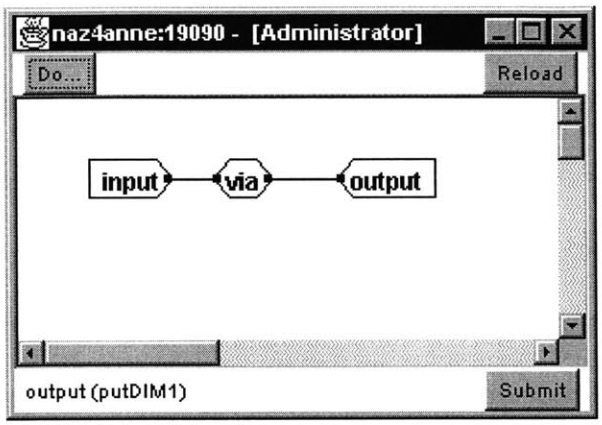


Figure 6-2: Picture of Bert

In iPAM, the data that crosses the connections is called a Product. There are two kinds of products: Basic and Compound. Basic products include a file, a string, or a URL. Compound products are lists of basic and compound products. The file product is very convenient for transferring XML files among agents.

In the design of the prototype, the JBI producers are implemented as input agents and the fuselets as handlers, with the exception of the car schedule fuselet, which is implemented as an output agent. The JBI consumers, car computers, are applications outside of iPAM. Though iPAM provided many of the features needed for this prototype, there was a major limitation: information channels in iPAM are static. The information administrator creates channels and once the system is started, the channels can only be changed by the

administrator. Therefore, the dynamic matching capability of the JBI could not be used to dynamically create information flows given a request. Instead, the channels that would have resulted from matching were created at start up. Regardless of this limitation, iPAM allowed the development of a prototype that demonstrates the major features of the system.

6.2.2 DirectionsFinder

DirectionsFinder is a route finding application that was developed as part of a problem set in the Laboratory in Software Engineering course at MIT during the Spring 2001 semester. The application includes a database of the Boston area as well as an application to find the best route between two locations. This application measures best in terms of least miles. The *DirectionsFinder* application was split into two pieces for the purpose of the CS system prototype. The database part of the system was used for the GIS producer and the directions finding module was used for the Route Finder. The application was further modified to meet the needs of the CS system.

The *DirectionsFinder* application views the world as consisting of a set of street segments that have a start and end geographical point, which consists of a latitude and a longitude. The segments have a name, a zip code, and a set of numbers associated with it. The implication of this model is that buildings cannot be referenced by their geographical point. Rather, they must be referenced by their address, which consists of a street number, a street name, and a zip code. The route finder application takes two addresses, finds the associated street segments, and determines the best route between the

two segments. Since the distance is only determined between two street segments, the distance returned is not exact. This slight imprecision was not an issue for the prototype.

The *DirectionsFinder* application has a couple of significant limitations. The first is that the directions are not completely accurate. The application assumes that each street is two- way and that you can turn left and right onto every street (i.e. there are no street dividers). Another limitation is that the application does not take into account whether a street is a highway or a local road when determining the best route. Therefore the application only returns the number of miles in a route, not the length of time for a trip, which is also important.

Despite these limitations, the *DirectionsFinder* application was chosen for its availability. Furthermore, the author of this paper was already familiar with the application and could readily adapt it for his uses. Since this system is only a prototype, the one-way street problem was overlooked. The application returns a route, which meets the requirements of the CS system. The amount of time for a trip was inferred from the length by assuming that the cars would be traveling at an average speed of 15 miles per hour. Though not completely accurate, it results in an approximate time, which is again sufficient for the CS system.

6.3 Fuselet Architecture

In order to enable rapid development of fuselets, a standard fuselet development framework was adopted. The framework consists of three classes: a wrapper class, a

fuselet class, and a handler class. The wrapper class is the class registered with the iPAM application. It is the entry and exit point of the data. The wrapper is responsible for converting between iPAM products and XML files and passing the files to the fuselet class. The fuselet contains the main application logic. It contains the knowledge to transform inputs to an output. This transformation is aided by the fuselet's handlers. The fuselet has a handler for each type of input. When the fuselet receives an input file, it passes it off to the appropriate handler for local processing, such as reading the XML file into local data structures. The handler is then used to access the contents of the XML file.

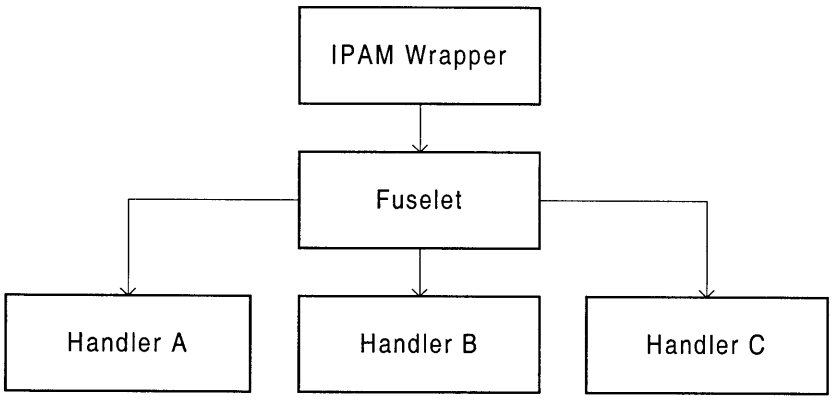


Figure 6-3. Fuselet Architecture

For example, in the cab service example there are two traffic report producers, CNN and MSNBC. The traffic report consolidator (TRC) takes information from both producers, and produces a consolidated traffic report. Using the fuselet framework, data enters the fuselet through the TRC wrapper. The XML file is retrieved from the iPAM product and is passed to the TRC fuselet. The TRC fuselet determines the source of the XML file and passes it to either the CNN handler or the MSNBC handler. Each handler contains the

knowledge to extract the important features from the traffic reports. The TRC fuselet uses the TRC handler methods to access the content of the XML files and perform the data consolidation. The fuselet then returns the resultant file to the TRC wrapper. The TRC wrapper converts the file to an iPAM product to be sent along the information channel.

This design separates the local and global processing that occurs in the fuselet. If changes are made to one of the inputs, only the input handler needs to be changed. The fuselet does not have to be changed allowing modifications to be made easily. Furthermore, for fuselets whose application logic does not depend on the type of the input, additional inputs can be added without changing the fuselet by simply adding the appropriate handler. An example of this is the location folder where another page can be added to the folder simply by connecting another input and adding a handler for that input. An even better example is the traffic reports consolidator, which can take any number of traffic report inputs without any modifications to the module.

6.4 Modules

The following is a description of the implementation of the modules that were prototyped. The XML document type definition for each module is located in the appendix.

6.4.1 Producers

GIS

The GIS producer produces a set of street segments. It has a database property that specifies the location of the tiger database and a zip code property that is used for filtering. Only the street segments with zip codes listed in the zip code property are returned. If the zip code property is empty, then all the street segments in the database are returned. Several thousand street segments are generated from this producer. Writing all of these segments to an XML file is time consuming, so caching is used to save on time and space. This module uses the street segment generator from the DirectionsFinder application. The GIS producer is implemented as a synchronous iPAM input agent.

MSNBC and CNN Traffic Reports

These producers produce traffic reports at the granularity of street names. The modules are asynchronous input agents. At start up, they load traffic reports from a comma-delimited file, which is specified from an iPAM property. There is a graphical user interface (GUI) that allows the user to specify new traffic reports. Once a new report is committed from the GUI, the agent is activated and pushes the report along the information channel. The modules have three properties. The first specifies the file to load reports from. The second specifies whether to display the GUI. The final property says whether or not to reload the traffic report file the next time the agent is run.

Images

The images producer is an asynchronous input agent that produces images. The module has a location property, which is used to select the images to output. There is also a count property and a rate property. The module is initially loaded with images from a comma-delimited file. The count property says how many images to initially output for the location. The rate property specifies how often to input new images into the system

Location Information

This producer produces location data that includes address, location name, and pickup points. A pickup point is a specific point at a location where a car can make a pickup. For example, a corner house may have two pickup points: the front door and the side door. The location module has one property that specifies the location(s) for which the producer produces information. This module is a synchronous input agent. The location data is loaded at startup from a comma-delimited file and remains the same throughout the running of the system.

Car Information

The car information module produces data about cars. It is a synchronous input agent that has no properties. The data is loaded at startup from a comma-delimited file and does not change. Car data consists of the car type (limousine or sedan) and the capacity of the car.

Scheduled Pickups

This producer outputs the pickups that have been scheduled for the day. The data are loaded at the start of the program from a comma-delimited file. It is a synchronous input agent and has no properties.

Sports Scores

The sports producer is an asynchronous input agent that produces sports scores. It has one property, which is the rate at which to produce new scores. The scores produced are generated randomly.

6.4.2 Fuselets

Traffic Reports Consolidator

This fuselet has two inputs, the MSNBC traffic report and the CNN traffic report. It is capable of taking more traffic report inputs. It is a no schedule handler without any properties. The fuselet resolves conflicting traffic reports by choosing the report with the later timestamp, breaking ties arbitrarily.

Route Finder

The route finder fuselet is a no schedule handler. It has two inputs, the GIS producer and the traffic reports consolidator. The traffic reports are used to filter out street segments so that they are not used in finding directions. Generating a map from street segments is very time consuming, so caching is used to avoid redundant computations.

Location Folder

The location folder has 3 inputs: images, location information, and route finder. The fuselet outputs a list of pages, one for each input. The handler for each input is responsible for generating the input's page. This module is a no schedule handler and has no properties. The module can support less than three inputs without any modifications to the module.

Car Planner

The car planner fuselet transforms the scheduled pickups into a car schedule. The car information module is used to retrieve all the cars in the system and their respective types and capacities. For simplicity, the module assumes that the time between making a pickup and arriving at the next pickup location is less than 90 minutes. The scheduling algorithm used is a greedy algorithm. It tries to schedule one car with as much work as possible before moving on to the next car. The car planner is a no schedule handler with zero properties.

Car Allocator

The car allocator fuselet receives requests for new pickups, determines which car should service the request, and sends an updated car plan to the car schedule. Its inputs are the car information and scheduled pickups module. The algorithm used to allocate a car chooses the first available car that can service a request. A graphical user interface is used to make a new pickup request. The user specifies the time, pickup location,

destination location, and number of passengers. The car allocator fuselet is a no schedule handler with one property to specify whether or not to display the GUI.

Car Schedule

The car schedule fuselet delivers each car its schedule for the day. It has three inputs: the car planner, the location folder, and sports scores. It takes the car plan and matches it with the location folders for the pickup locations. Presentation information is added along with the sports scores. Then each car is sent its schedule. The car schedule fuselet is implemented as an iPAM output agent.

Image Converter

The Image Converter converts jpeg images to gif images. The input to this fuselet is the image producer. The implementation converts a file named x.jpg to a file named x.gif. This fuselet is a no schedule handler with zero properties.

Location Filter

The Location Filter filters out a field from the location information. The field to be filtered is specified in the fuselets only property. The location filter is a no schedule handler.

6.4.3 *Consumers*

Cars

A web browser is used to represent the car computers. Each browser is pointed at an html file that is the driver's home page. The links on the homepage are updated when new information is made available that is of interest to the car.

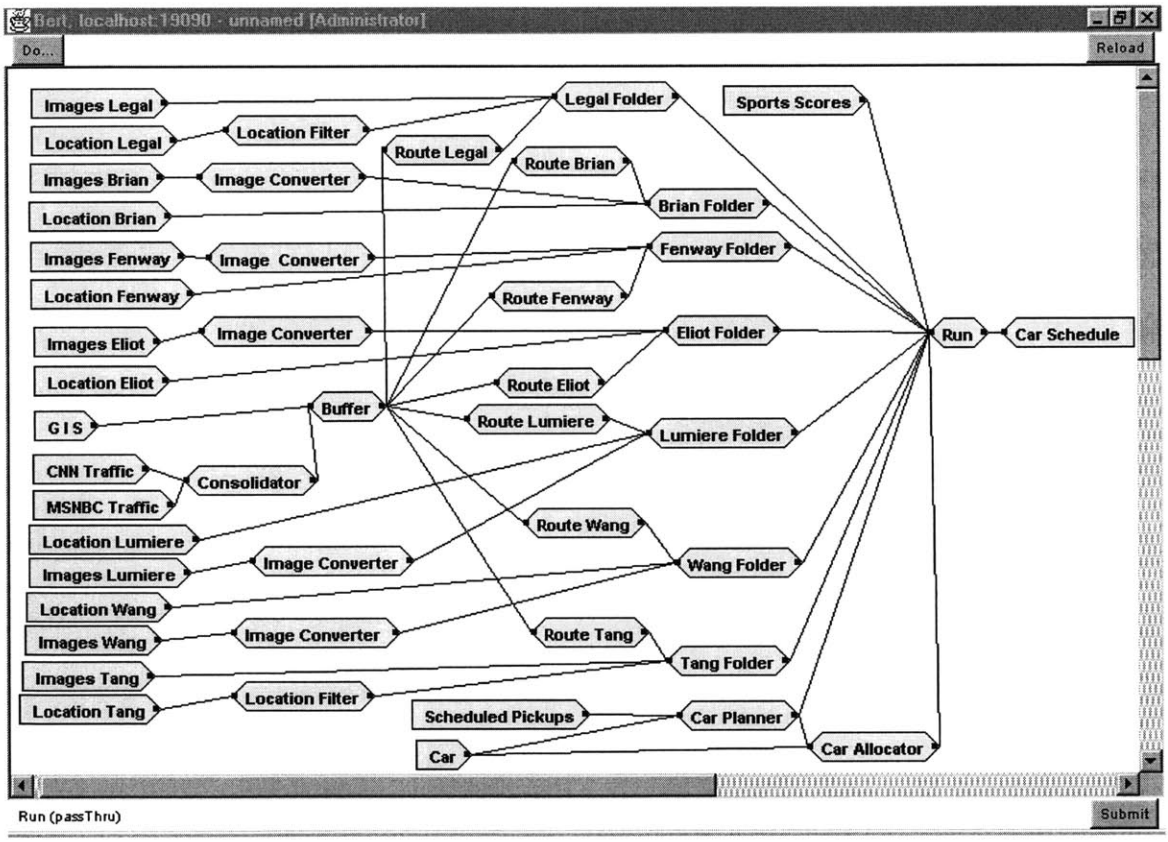


Figure 6-4. Car Service System

6.5 *Scenario Dynamics*

Figure 6-4 shows the set up of the iPAM agents. Since iPAM agents have a single output, modules that require different outputs for the different information channels were replicated in the network. For example, there are seven route finder modules. Each one

has its start and destination property set to match the location folder it is fed into. Other components such as traffic reports and sports scores give the same output for each information flow, so they are not replicated. The one output is fed into each information flow that needs it. There are two additional agents in the network that were not mentioned above, the Buffer and Run agent. Both agents simply pass whatever is at their input to the output. The Buffer is a no schedule handler used to reduce the number of edges in the graph. The Run agent is a schedule handler that is used to start the system. When it is activated, it pulls on all of its inputs causing a domino effect that ends at the producers. This represents the start of the day. The order in which the iPAM executive runs the agents is not fully specified. There is only a partial order defined by the input/output constraints of the information channel.

The channels that bring the location folders to the car schedule module are located in the upper half of Figure 6-4. They will be referred to as the location folder channels. The traffic report producers reads in a traffic report file, which it uses as its initial set of traffic reports. It then writes them to an XML file and passes it to the traffic reports consolidator. The traffic reports consolidator receives the files from the MSNBC and CNN traffic report modules and consolidates them into a singe traffic reports file. This is then sent to the route finder. The route finder also receives an XML file specifying street segments. The route finder parses the street segment file and constructs a map of the area. Segments that have traffic on them are not placed in the map. The route finder finds the best route between the start and destination address. The addresses are specified by the start and destination property of the agent. The best route along with its length is

sent to the location folder. The location information and images modules pass the location folder data about the location that is specified in their properties. The location folder aggregates all of this information and sends it on to the car schedule fuselet.

The location folders channels are slightly different for the Boston and Cambridge car companies. Boston car computers can read only gif images, but the image producer only produces jpeg images. Therefore, an image converter fuselet is placed in between the image producer and the location folders that are meant for Boston cars. The fuselet converts jpeg images to gif images. Another difference is that the Cambridge car company does not want the cars to know about the function of a pickup location. Therefore, a location filter is placed in between the location information and the location folders that are intended for the Cambridge cars. The filter removes the function tag from the XML file.

The lower half of Figure 6-4 is the information channel that determines the car schedules. The scheduled pickups for the day along with the car information are sent to the car planner. The car planner uses a greedy algorithm to match cars to pickups. The schedule for each car is then sent to the car schedule fuselet.

The sports score producers feeds directly into the car schedule fuselet. It sends the car schedule current scores.

The car schedule fuselet is responsible for sending each car a set of web pages through which drivers can view their schedules. The first step required is to match up the location folders with the pickups. This is done by indexing the location folders on their start and end addresses, which is determined from the route that the folder contains. The location folder for a pickup can be found by looking at the pickup location and the destination location of a pickup. Once a pickup is matched with a location folder, an html page is made for each pickup with links to location folder items such as images and directions. A car homepage is then made with links to each of its assigned pickups. The pickup assignments are retrieved from the plan received from the car planner. A link to the sports scores is also attached to each homepage. The homepage for car "x" is written to carx.html. The driver of car x can view his or her schedule by going to his or her page. It is updated whenever relevant information is made available. A more realistic implementation of the car schedule would have been a servlet. However, writing to html files on the local file system is sufficient for the purposes of the prototype and avoids the complexity that comes with servlets.

There are four operations that can cause an html page to be updated. The first is if the sports scores module sends out a new set of scores. This information is common to all drivers, so each driver's homepage has the sports score link updated. The second operation is a new image being made available. This causes the image module to send out a new list of images. The new set of images is pushed along the information channel through the location folder to the car schedule. The car schedule uses the new images to make a new html page and changes the links on the appropriate pickup pages to point to

the new image page. The third operation is a new pickup being entered into the system through the car allocator. This causes the car allocator to assign the pickup to a car and send an updated car plan to the car schedule. The car schedule then updates the car's homepage to reflect this new pickup. Finally, a change in the prediction of the traffic report aggregator can modify previously computed routes. If a new traffic report is made, this information gets pushed to the route finder. The route finder uses any new information to rebuild the map and find a new best route. This route is then pushed to the location folder, which then pushes it to the car scheduler. The car scheduler uses the new information to update the previous route pages.

The images module for the Wang Center and the Tang Hall were set up to produce new images every 60 seconds. The sports score module was set up to produce new sports scores every 90 seconds. Whenever the image or sports score module pushed data onto their information channels, the schedules for the effected cars were updated. Three traffic reports were added to the system from the traffic report GUI. The first specified traffic on Milestone Road. Milestone Road is used by every route that was computed, so each route finder rebuilt its map and found a new route that did not use Milestone Rd. This new route was pushed to the car schedule fuselet and the directions for all the cars changed. The next traffic report was for Memorial Drive. Since none of the routes use Memorial Drive, no direction pages were updated. Finally, a traffic report for Hummock Pond Rd was added. Only one route used that street. Therefore, only one route finder recomputed directions and forwarded it to the car schedule.

Chapter 7. Evaluation

In Chapter 2, the status of mission critical systems was presented. A vision for future systems was described that included vendor independence, hardware/OS independence, and minimal component dependencies. In this chapter, we evaluate the design and implementation of the car service (CS) system in terms of how well it achieves this vision. The implementation of the prototype brought new insight. These insights are also shared.

7.1 Was the vision achieved?

The CS system is written entirely in Java. Therefore, it is both hardware and operating system independent. The system does depend on a few software vendors, but migrating it to products from other vendors would not be difficult. The GIS producer depends on the DirectionsFinder street segment generator and the tiger database. A new database could be used in place of the system, as long as a method is devised to convert the new database into a list of street segments. Similarly, the Route Finder fuselet depends on the DirectionsFinder algorithm, which uses street segments. However, a different route finding algorithm could be used without any problems. Both of the changes would be local to the module(s) involved, and would not affect the rest of the system.

The CS system also depends on the iPAM environment. The system was implemented so that it would not be difficult to move the components to another distributed component architecture. Each module has a separation between the business logic and the iPAM specific functionality. The business logic is wrapped in an iPAM wrapper class whose

only responsibility is to communicate with the iPAM system. Therefore, to move the business logic to another distributed architecture, only a wrapper class would need to be implemented to communicate inside the new architecture. Very few changes would be necessary to the business logic. For example, to use this system in an application server with Enterprise Java Beans (EJB) as the component architecture, only the EJB specific classes would need to be built. These EJB classes would then use the old business logic to perform the work.

The major part of the vision for mission critical systems is the ability to easily add new modules and modify existing ones. A significant hindrance to easy addition and modification of modules is the existence of dependencies. Therefore, throughout the design of the CS system, minimizing dependencies was a driving force in many decisions. A good example of achieving minimal dependency is the location folder. The car schedule fuselet needs to obtain location information and images for every pickup location. It also needs directions for each trip and maps of the area for each trip. All of this information is located inside the producers. One option would be for the car schedule to retrieve the information from each producer and manipulate the data itself to get the desired results. This would however have resulted in a large number of dependencies and a large amount of complexity for the car schedule fuselet. Instead, the car schedule fuselet retrieves most of this information from the location folder. The location folder is responsible for retrieving the location information, images, maps, etc. Not only does adding the location folder greatly reduce the number of dependencies for the car schedule, it also simplifies the fuselet. Instead of sorting through a bunch of

information about locations, the information arrives at the car schedule module presorted from the location folder. Each folder received from the location folder contains only information about a given location. This design pushes most of the module dependencies from the car schedule to the location folder. The car schedule still maintains the dependence on weather maps, because the weather data are for a general area rather than a specific location.

Now the location folder is responsible for aggregating all the information from the remaining producers. It communicates with the producer when all that is needed is the raw information. This is the case for location information and images. For information that first needs to be manipulated, a service is used to manipulate the data. Then the manipulated data are passed to the location folder. In instances where the data need to be acquired from multiple sources, this scheme reduces the number of dependencies for the location folder. This is true for determining the route, which uses four producers: road construction, geographic/mapping data (GIS), CNN traffic reports, and MSNBC traffic reports. For the map overlay fuselet, which uses only the weather reports and maps producer, there is no minimizing of dependencies, but it does make the location folder less complex allowing the location folder to only be concerned with aggregating information.

By introducing additional modules into the system, the dependencies for each module has been reduced to a manageable amount, thus lowering the complexity of each module. In addition, modifications and additions to the system can be made without much difficulty.

By breaking the system up into many modules that perform specific tasks, new modules can be easily added. The image converter is an example of a component that can be easily added to the system that enhances the functionality of the system. It is an example of the formatting fuselet described in chapter 4. The image converter converts image files to a specified format. It then outputs the new image data. The ability to seamlessly add this component greatly increases the usability of the system. Assume for example, new consumers enter the system that can only read tiff images. By simply adding an image converter after the image producer for every information channel that ends at one of the new consumers, the new clients of the system can use all of the existing services. Another example of easy addition of modules is traffic reports. A new traffic report module can be entered into the system by merely linking it up with the traffic reports consolidator. Once this is done, the whole system can make use of this new module without any changes.

Completely new information channels can also be added easily. Each component is created without knowledge of which components use it. Therefore, new information channels can be made without altering any preexisting channels. For example, let's say a new requirement comes from the car drivers that state they want to know about all the gas stations along a route in case they are low on gas. By using the route finder, location information, and the GIS producer, a fuselet can be constructed that provides this service. A new information channel is made and no existing fuselets are modified.

Many modifications can also be made to the CS system without affecting much of the rest of the system. The system was broken up into many independent modules. As long as the output of these modules remains constant, modifications will only affect the module being changed. Examples of this type of change are using a more efficient route finding algorithm in the route finder or improving the scheduling algorithm in the car planner. If, however, the modification changes the output format, all the modules that use the modified module will have to be changed. Since receiving the data was kept separate from the core business logic, only the input parsing of client applications will need to be modified. On the other hand, if the original modification changed not only the output format, but also the output content, then major changes may be necessary for clients of the module. In the CS system, no module has more than four clients. Keeping the fan out of modules small limits the ripple effect of the modifications.

7.2 Performance/Scalability

In this section, the performance and scalability of the CS system is evaluated. All the numbers mentioned in this section are from running the system on a Sun Ultra Sparc 5.

Most of the components in the CS system only take a few seconds to run. The main bottleneck of the system is computing directions between two locations. The route finder is a computation intensive application that takes about 27 seconds to complete. This time is broken up into 7 seconds to parse the street segments file, 18 seconds to build the map, and 2 seconds to find the directions. Caching the previous directions was used to reduce the time to find a route. The only reason for a route to change is if there is new traffic

data. If the new traffic data contains a street that was used in the directions previously computed, the map must be rebuilt and the directions must be recomputed. If this is not the case, then the directions that have already been computed can be returned. This can reduce the time to find directions by almost 70%. This caching greatly increased the performance of the system after start up.

The initial set up time for the system was about four minutes. Three minutes of this time was spent computing directions. Efforts to increase the efficiency of computing the directions are described below. After start up, changes to the system such as new pickups and images were completed in just several seconds.

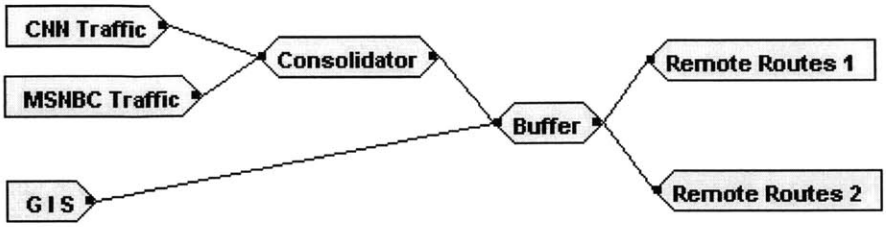
To assess how well the system scaled, a number of parameters were increased. First, the number of cars, locations, images, and traffic reports were increased. The impact on performance was negligible. However when the number of pickups was increased, the performance impact was significant. The reason for the slow down is the route finder. For each pickup, a route has to be computed that gives directions between the pickup location and the destination location. Therefore, as the number of pickups is increased, the number of directions that must be computed increases. As mentioned above, direction finding is computationally intensive and is the bottleneck in the car service system.

In order to minimize the time to compute the car schedules, two techniques were used. The first technique was to share one instance of the DirectionsFinder class among all

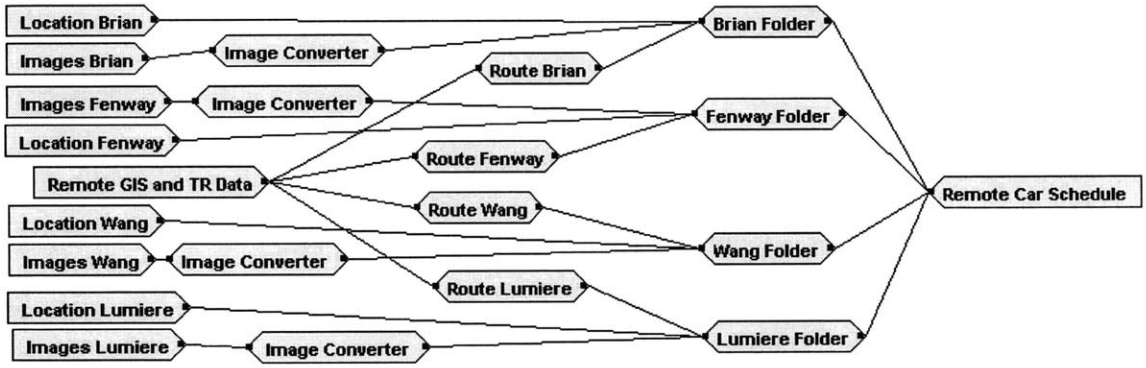
route finder components. The DirectionsFinder class is responsible for building the map. Since each route finder had its own DirectionsFinder, multiple maps were being built. There is no difference between these maps when the system is initially started. Therefore, sharing the maps between the route finding modules would eliminate redundant computation. Sharing maps amongst the route finder components reduced the start up time of the system by 34%. Sharing the modules is representative of how the CS system would be implemented in real life. All of the route finder components would be located in a route finder server, which would service the requests for directions.

A second technique used to alleviate the bottleneck in the route finder components was dividing the car service system amongst multiple computers. Four Sun Ultra Sparc 5 machines were used. Figure 7-1 shows how the system components were assigned to each machine. This set up uses seven pickups. Machine 1 is responsible for delivering the GIS and traffic data to the route finder modules. Machines 2 and 3 are responsible for making the location folders. Machine 2 is assigned three folders and machine 3 is assigned four folders. Machine 4 has the task of computing the car schedules and delivering them to each driver. In order to transfer information among machines, the iPAM serialization input and output agent was used. The output agent takes an iPAM product, serializes it to a Java object stream and sends it over the network connection to the destination machine. The input agent receives a serialized iPAM product over the network and pushes it onto the information channel.

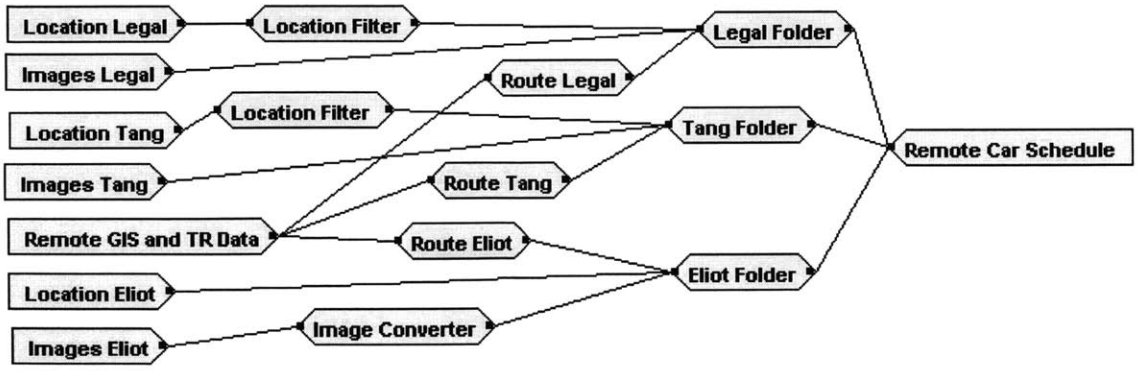
Machine 1



Machine 2



Machine 3



Machine 4

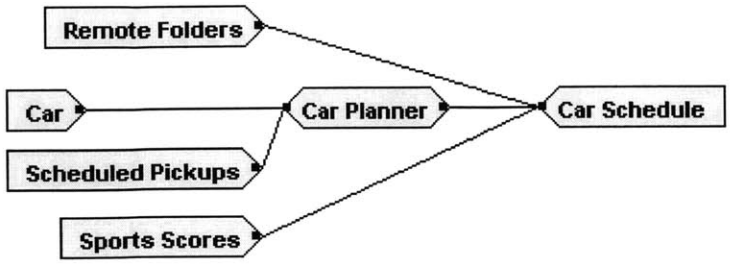


Figure 7-1. Car service system distributed on four machines

The distributed version of the car service system also completed the initial startup about 34% faster than the non-distributed version. The speed up came from the reduced time in computing routes. By spreading the computation of the directions over two computers, the total time to compute directions was decreased. If the directions finding were spread across more computers, the speed up would be even greater.

Both techniques reduced the start up time for the car service system by a significant amount. The first technique of sharing the DirectionsFinder works well during start up because each route finder uses the same map. However, it would not have as large an effect if the route finders were using different maps. For example, if some route finders were using maps of Boston and others were using maps of Cambridge, the same map could not be shared. The second technique of distributing the components still works well in this case, since each route finder has its own copy of the DirectionsFinder. When

both techniques were used together, the start up time was reduced to only 90 seconds, which represents an improvement of over 60%.

The rate at which the sports scores and the images are created were also increased. This did not have a significant effect on the performance of the system.

Finally, the size of the map was increased. The map used in the prototype was only a small subset of the Cambridge area. When the size of the map was increased to include most of the Boston and Cambridge area, the performance of the system decreased significantly. Once again, the problem was in the route finder. The first problem was reading in the street segment file. There are over sixty thousand segments in the file. Parsing this file and constructing the street segment objects took several hours to complete. After the file is parsed, building the map takes 14 minutes. After this is done, finding the route takes about ten seconds to complete.

The large start up time for the CS system with the larger map is unacceptable. It appears that as the map grows in size, XML is not the appropriate format with which to translate the data. It would be much better if the data were transferred in some binary format. Transferring the data as a serialized Java object was tried. It took only 7 minutes to read the serialized objects, as opposed to the hours it took to read the XML objects. This reduced the start up time tremendously.

7.3 Lessons Learned

During implementation, several design problems became apparent. In this section, several of these are presented.

7.3.1 Generic Java classes for each data type

Fuselets receive their inputs in the form of an XML file. While it is possible to perform whatever computation is necessary directly from the file, it is often easier to read the file into a class that represents the file. For example, the traffic reports producer produces a list of traffic reports, using the traffic report schema. The traffic report consolidator receives several of these and aggregates the reports. The implementation of the consolidator uses a traffic report class to represent each report. It then uses the class to access the contents of the report such as the road and timestamp. The route finder also receives traffic reports and uses a traffic report class to represent each report to aid in determining the best route given the reports. Initially, the idea in the design of this project was that each module would be completely independent. However, having a generic traffic report class would have been useful and would have saved some duplicate effort. The same situation arose with the other data types such as cars, locations, and pickups. In future implementations, it would be better for each data type to have a generic Java class that represents the type and to make it accessible to application developers. Application developers will then have the option of using this class to store the objects in the XML file, rather than writing their own class from scratch.

7.3.2 Formatting and Filtering Fuselets

In the design of the system, it was stated that formatting and filtering fuselets could be placed anywhere in the system and the rest of the system would work without modification. During implementation it was found that this is only true for filtering fuselets if the schema supports it. The original schema for location information specified that every field of the location data type had to be present. Therefore, the location folder expected each field to be present. However, when a filter was placed after the location information removing a field, the system broke down. Fixing the system required two changes. The first was to make all of the fields in the location information schema optional. The second was to change the source field of the filtered information to be the original module, location information. The latter change was necessary because the location folder determined its input based on the source. When the source is Location Information, it would route the data to the location handler. However, when the source was Location Filter, the location folder did not know how to handle it.

The problem with filtering brought up the notion of pretending to be another module. For the location folder to work without any changes, the location filter had to pretend to be the location information producer. Another solution would be for each schema in the system to not only have a source field, but also a service field. The source field would be the name of the module and the service field would be the name of the service that module provides. Using this scheme, the location information and location filter would each use its own name as the source and set the service field to location. This schema also supports having multiple sources of the same data.

7.3.3 Tradeoffs in minimizing dependencies

Another issue occurred when trying to minimize dependencies in the system. In the prototype, the goal was for the car schedule to only use the car planner and the location folder. For the car schedule to provide each driver with his or her schedule, the module must match the location folders to the drivers that need the folder. To do this, intermediate mappings were made from drivers to pickups and then pickups to location folders. A problem occurred, however, when trying to map pickups to location folders. Pickups only contained the location name and id of the pickup and destination location. The location folder only contained the addresses of the source and destination location, because the route finder only works with addresses. Therefore there was no way to find the location folder for a pickup since there was no way to map addresses to locations.

Several solutions were available at this point. One was to make the car schedule use the location information to obtain the mapping between addresses and location ids. Another was to have the car plan not only contain the location name and id, but also the address for each location. The latter solution was chosen because it would avoid adding another dependency in the system. However, it was done at the expense of increasing the size of each pickup file. Pickups now included the addresses along with the location names and ids. An idea that was not explored is using a filtering fuselet. So far in the paper, filtering fuselets have been thought of as removing information. However, there is no reason that they cannot add information. A filtering fuselet could have been used after the car planner to add address information to the car plan. This would have allowed the

Scheduled Pickup producer and Car Planner fuselet to be untouched. This option adds dependencies much like the option of allowing the car schedule fuselet to use the location information. In this case however, the dependency is being placed on a relatively simple, non-crucial module in the system. If a change occurs in the location information producer, the filtering fuselet can be updated much more easily than the car schedule fuselet.

7.3.4 Presentation

In the description of the design in chapter 4, it was stated that modules that deal with presentation do not perform any other function. This idea was not strictly adhered to in the prototype. During the design, the process of displaying the information from the car planner and location folder was perceived as trivial. However, during implementation it was discovered that matching car plans to location folders was not simple and should have been the responsibility of another fuselet. In the revised design, the location folder and car plans would be sent to a fuselet. This fuselet would match car plans to location folders and send a file containing these to the car schedule fuselet. The car schedule fuselet would then format the file for whatever display format it supports.

Chapter 8. Further Research

In this chapter, several areas where further work is needed are explained. These areas range from system level design to the implementation of individual components.

8.1 Joint Battlespace Infosphere

Though the system was designed for use in the Joint Battlespace Infosphere (JBI), the prototype was implemented on iPAM because the JBI is still in development. When a version of the JBI is made available, the CS system modules that have been implemented should be moved to the JBI architecture. This would require writing a wrapper for each module that handled communicating with the JBI. Then performance measurements can be taken and scalability tests can be performed.

8.2 Specifying Publications and Subscriptions

Throughout this paper, it has been assumed that the information channels can be set up. The mechanism to arrange the information channel was overlooked. One needs to study how these information channels will be set up. In the JBI, information channels are determined through a set of publications and subscriptions. Figure 8-1 shows part of an information channel that exists in the car service system prototype.

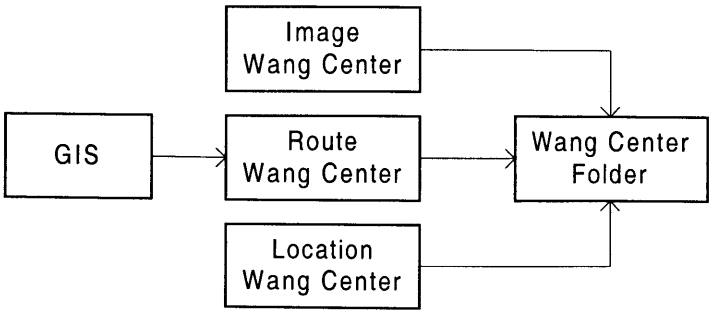


Figure 8-1.

For the JBI broker to be able to set up the information channel above, the GIS producer needs to publish that fact that it produces street segment data. The route finder needs to publish that it produces directions from the Wang Center to the Tang Hall. The location information and image producers need to publish that they produce location data for the Wang Center and images for the Wang Center, respectively. In addition to these publications, the route finder needs to subscribe to street segment data and the location folder fuselet needs to subscribe to directions from the Wang Center to the Tang Hall, to Wang Center location data, and Wang Center image data. Given these publications and subscriptions, the JBI broker can determine that the above information channel needs to be set up. Exactly how to formalize these publications and subscriptions in a language needs to be studied. There is work currently going on inside the MITRE corporation on developing a profile language to express publications and subscriptions.

8.3 Presentation

The majority of the work on this system was on channeling the information from the producers to the clients in a modular and extensible way. Not much attention was paid to displaying this data. Further work on presenting the data could greatly enhance the usability of the system. One area that may be of interest is linking between pages. For example, if a user were on the page that displays the directions, it would be useful to be able to click on the name of the source location, which causes the location information to then appear. Another useful feature would be a link from the directions page to the route finder fuselet, which provides a page on which the user can query for directions. Another

area for further research relates to customizing the schedule pages for each user. A trivial example is customizing the initial homepage so that the trips are sorted in descending order, rather than ascending order.

8.4 Feedback

Throughout the prototype of the CS system, it was assumed that the information in the system was already present. Changes to information in the system were made through the traffic reports, images, car allocator, and sports module. In the design of the overall system however, a more dynamic and interactive environment exists where there is feedback into the system from the driver and the monitoring modules. Exactly how the driver and monitoring modules input information into the system and how the system reacts to this new information is an area for further research.

8.5 Optimal distribution and replication of modules

A major area of further research is the optimal distribution and replication of modules. There are two driving forces behind these choices. The first is the speed of the system. Modules that interact often should not be placed far apart. However, having too many modules on the same machine may negatively impact performance. Also, clients of the system should not be far from the system, else network bandwidth might become a bottleneck. Since clients may be very apart, techniques such as replicating modules may be necessary so that all clients are near the modules they access.

The other driving force is availability. It is possible that a computer may go down. If this computer held the only version of several modules, then the CS system itself may go down. This single point of failure is not acceptable. Therefore, modules should be replicated. The replicated modules should not be close together in case the reason the computer went down is because a thunderstorm happened to knock out electricity on that block. However, managing distributed replicated components is complex, since the replicated modules need to be kept in sync. Further study is required to determine the exact number and location of the replicated components.

Chapter 9: Conclusion

The goal for the design of the car service (CS) system was to produce a system that was portable among hardware and software vendors, and also allowed easy addition and modification of system components. The design of the CS system was successful in this respect. Other than the JBI architecture upon which the system is built, there are no requirements on software or hardware vendors. In addition, the system was broken up into many distributed components, each with a specific role. Having many specific components decouples the individual jobs that need be done in performing a task, facilitating modifications to the individual jobs. Also, through the use of the publish/subscribe paradigm, additions to the CS system are relatively simple. New information flows can be set up by simply adding the necessary modules and making the appropriate publications and subscriptions. No changes to existing modules are necessary. In some cases such as filtering, existing flows can be altered without making any changes to existing components as well.

The prototype of the CS system demonstrated many of the ideas present in the design. The system was written entirely in Java, so it can be run on a variety of operating systems. Also, aside from iPAM, the system is not dependent on any software vendor. Many major changes can be made to the system by only altering one module. No other modules need to be altered. In addition, new information flows can be easily set up by adding the new components and connecting them together. These flows can make use of existing components without altering those components. During implementation, several issues arose from which a few lessons were learned. First, as the number of routes that

need to be computed increases, the route finding should be distributed across several computers. Information should be shared between route finder modules whenever possible, and caching should be used. Another lesson is that if the size of the data to be transferred is very large, it may be necessary to transfer the file in some binary format, rather than XML to improve the performance of the system. There is however a tradeoff here. By transferring data in a binary format, the coupling between the modules tightens. If speed is important, then tight coupling between components may be necessary. In the CS system, this tight coupling was only needed for two modules. Finally, the data schemas need to be carefully defined to allow transparent filtering and to minimize the dependencies among components. If the schemas are not defined appropriately, filtering may require changes to several modules and the dependencies among components may increase.

One purpose of this research was to explore using the Joint Battlespace Infosphere (JBI) as the underlying architecture for mission critical systems. The results of the prototype showed that an information channel approach can lead to a portable, modular, and extensible system. Also, through clever implementation, the system can scale to support a large amount of data. The question that remains unanswered is how efficiently can the information channels be set up. This question is one that can not be answered until an implementation of the JBI is available. If, however, the information channels described in this paper can be formed and maintained efficiently, the JBI will be a suitable architecture for mission critical systems.

The CS system was devised as an analogy to TBMCS. The CS system solves many of the current problems in TBMCS, such as many interconnected modules and vendor dependence. It allows changes to be made to the system without affecting many individual components. The design of the system also has a low barrier to entry. A variety of clients can enter the system without much work. Data is available in many formats and new formats can be easily added through the use of a formatting or filtering fuselet. The CS system can be mapped back to TMBCS, making its design a suitable choice for future versions of TBMCS.

This paper started out by discussing the status of mission critical systems. A vision was stated for the future of these systems that included being browser enabled, vendor and OS independent, and extensible. To examine methods to achieve this vision, TBMCS was chosen as the focus of study, because it is representative of many mission critical systems today. The JBI was chosen as the architecture upon which to base the redesign of TBMCS. Due to the sensitivity of data processed by TBMCS, a car service system analogy was used to prototype a proposed architecture for TBMCS based on the JBI. The car service system devised is very similar to TBMCS and the ideas present in the CS system also can be mapped to TBMCS.

The results of the research showed that the CS system meets or exceeds the requirements laid out for future mission critical systems. Since the CS system maps to TBMCS, the CS system design, which is based on the JBI, is a suitable design for TBMCS. Current mission critical systems share many of the problems with TBMCS such as the complexity

caused from module dependencies and inability to move applications to different operating systems and vendors. In addition, many mission critical systems can be thought of as channeling information from a data source to clients of the data. In channeling this data, the information is combined, analyzed, and formatted. Therefore the results of the research presented in this paper can be applied to many mission critical systems. The design of the CS system shows that the JBI can be used to develop a system that achieves the vision for mission critical systems.

References

[1] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, D. Sturman. An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems. 19th IEEE International Conference Distributed Computing Systems, 1999.

[2] G. Banavar, M. Kaplan, K. Shaw, R. Strom, D. Sturman, W. Tao. Information Flow Based Event Distribution Middleware. 19th IEEE International Conference on Distributed Computing Systems Workshops on Electronic Commerce and Web-based Applications/Middleware, 1999.

[3] T. Berners-Lee. Information Management: A Proposal. http://www.w3.org/History/1989/proposal.html, 1989.

[4] K. Betz. A Scalable Stock Web Service. Proceedings of the 2000 International Workshops on Parallel Processing, 2000.

[5] M. B. Blake. Rule-Driven Coordination Agents: A Self-Configurable Agent Architecture for Distributed Control. 5th International Symposium on Autonomous Decentralized Systems, 2001.

[6] M. B. Blake, P. Liguori. An Automated Client-Driven Approach to Data Extraction using an Autonomous Decentralized Architecture. 5th International Symposium on Autonomous Decentralized Systems, 2001.

[7] C. Bornhovd, M. Cilia, C. Liebig, A. Buchmann. An Infrastructure for Meta-Auctions. Second International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems, 2000.

[8] T. Bray, J. Paoli, C. Sperberg, E. Maler. Extensible Markup Language (XML) 1.0 (Second Edition). http://www.w3.org/TR/2000/REC-xml-20001006

[9] M. Franklin, Z. Zdonik. A Framework for Scalable Dissemination-Based Systems. Proceedings of OOPSLA, 1997.

[10] H. Hermansson, M. Johansson, L. Lundberg. Seventh Asia-Pacific Software Engineering Conference, 2000.

[11] C. Li, J. Smith, R. Mohan, Y. Chang, B. Topol, J. Hind, L. Yongcheng. Distributed Application Service for Internet information portal. The 2000 IEEE International Symposium on Circuits and Systems, 2000

[12] B. Liskov, J. Guttag. Program Development in Java. Addison-Wesley, Boston, 2001.

[13] M. Morrison, et al. XML Unleashed. Sams Publishing, 2000.

[14] B. Oki, M. Pfluegl, A. Siegel, D. Skeen. The Information Bus – An Architecture for Extensible Distributed Systems. Proceedings of the 14th ACM Symposium on Operating Systems Principles, December 1993.

[15] C. Sperberg-McQueen, L. Burnard. A Gentle Introduction to SGML. http://www.uic.edu/orgs/tei/sgml/teip3sg/

[16] Talarian Corportation. Publish & Subscribe: Middleware with a Mission. http://www.talarian.com/products/smartsockets/whitepaper.shtml, 1998.

[17] United States Air Force Scientific Advisory Board. Report on Building the Joint Battlespace Infosphere Volume 1: Summary, 2000. http://www.sab.hq.af.mil/Archives/

[18] T. Yan, H. Garcia-Molina. SIFT – A Tool for Wide-area Information Dissemination. Advanced Computing Systems Association (USENIX) Technical Conference Proceeding, 1995.

[19] N. Zhong, J. Liu, Y. Yao, S. Ohsuga. Web Intelligence. The 24th Annual International Computer Software and Applications Conference, 2000.

[20] http://www.cda.mews.org

[21] http://java.sun.com/products/ejb

[22] http://www.bea.com

[23] http://www.corba.org

[24] http://www.ibm.com/websphere

[25] http://www.netscape.com

[26] http://www.w3c.org/MarkUp/SGML/

[27] http://www.w3c.org/TR/SOAP

[28] http://www.w3c.org/xml

Appendix

Document Type Definitions for Car Service System data types

CarPlans.dtd

```
<!ELEMENT carplans (module, carplan*)>

<!ELEMENT module (#PCDATA)>
<!ELEMENT carplan (carid, pickup*)>

<!ELEMENT carid (#PCDATA)>
<!ELEMENT pickup (id, srcloc, destloc, time, passengertype,
                  numofpassengers)>

<!ELEMENT id (#PCDATA)>
<!ELEMENT srcloc (locationid, locationname, address, pickuppt)>
<!ELEMENT destloc (locationid, locationname, address, pickuppt)>

<!ELEMENT address (number, name, zip)>
<!ELEMENT number (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT zip (#PCDATA)>

<!ELEMENT locationid (#PCDATA)>
<!ELEMENT locationname (#PCDATA)>
<!ELEMENT pickuppt (#PCDATA)>
<!ELEMENT time (#PCDATA)>
<!ELEMENT passengertype (#PCDATA)>
<!ELEMENT numofpassengers (#PCDATA)>
```

Cars.dtd

```
<!ELEMENT cars (module, car*)>

<!ELEMENT module (#PCDATA)>
<!ELEMENT car (id, company, cartype, capacity)>

<!ELEMENT id (#PCDATA)>
<!ELEMENT cartype (#PCDATA)>
<!ELEMENT capacity (#PCDATA)>
<!ELEMENT company (#PCDATA)>
```

Images.dtd

```
<!ELEMENT images (module image*)>

<!ELEMENT module (#PCDATA)>

<!ELEMENT image (id, title, location+, filename)>
<!ELEMENT location (locationid, locationname)>
<!ELEMENT locationid (#PCDATA)>
<!ELEMENT locationname (#PCDATA)>
<!ELEMENT filename (#PCDATA)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT id (#PCDATA)>
```

Location.dtd

```
<!ELEMENT locations (module, location*)>

<!ELEMENT location (id, name, address, function, capacity, hours,
pickuppt+)>

<!ELEMENT module (#PCDATA)>

<!ELEMENT address (number, name, zip)>
<!ELEMENT id (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT function (#PCDATA)>
<!ELEMENT pickuppt (#PCDATA)>
<!ELEMENT capacity (#PCDATA)>
<!ELEMENT hours (#PCDATA)>

<!ELEMENT number (#PCDATA)>
<!ELEMENT zip (#PCDATA)>
```

LocationFolder.dtd

```
<!ELEMENT locationfolder (module, foldername, page*)>

<!ELEMENT module (#PCDATA)>
<!ELEMENT foldername (#PCDATA)>

<!ELEMENT page (pagename, (routes | location | images))+>

<!ELEMENT routes (start, end, route*)>
<!ELEMENT route (directions, length))>
<!ELEMENT start (number, name, zipcode)>
<!ELEMENT end (number, name, zipcode)>
<!ELEMENT number (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT zipcode (#PCDATA)>
<!ELEMENT directions (#PCDATA)>
<!ELEMENT length (#PCDATA)>


<!ELEMENT location (id, name, address, function, capacity, hours,
                    pickuppt+)>
<!ELEMENT id (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT address (number, name, zip)>
<!ELEMENT zip (#PCDATA)>

<!ELEMENT function (#PCDATA)>
<!ELEMENT pickuppt (#PCDATA)>
<!ELEMENT capacity (#PCDATA)>
<!ELEMENT hours (#PCDATA)>

<!ELEMENT images (image*)>
<!ELEMENT image (id, title, filename)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT filename (#PCDATA)>
```

Pickups.dtd

```
<!ELEMENT pickups (module, pickup*)>

<!ELEMENT module (#PCDATA)>
<!ELEMENT pickup (id, srcloc, destloc, time, passengertype,
                  numofpassengers)>

<!ELEMENT id (#PCDATA)>

<!ELEMENT srcloc (locationid, locationname, address, pickuppt)>
<!ELEMENT destloc (locationid, locationname, address, pickuppt)>

<!ELEMENT address (number, name, zip)>
<!ELEMENT number (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT zip (#PCDATA)>

<!ELEMENT locationid (#PCDATA)>
<!ELEMENT locationname (#PCDATA)>
<!ELEMENT pickuppt (#PCDATA)>
<!ELEMENT time (#PCDATA)>
<!ELEMENT passengertype (#PCDATA)>
<!ELEMENT numofpassengers (#PCDATA)>
```

Route.dtd

```
<!ELEMENT routes (module, start, end, route*)>

<!ELEMENT module (#PCDATA)>
<!ELEMENT start (number, name, zipcode)>
<!ELEMENT end (number, name, zipcode)>
<!ELEMENT route (directions, length)>

<!ELEMENT number (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT zipcode (#PCDATA)>
<!ELEMENT directions (#PCDATA)>
<!ELEMENT length (#PCDATA)>
```

Segments.dtd

```
<!ELEMENT streetsegments (database, zipcode*, segments)>

<!ELEMENT database (#PCDATA)>
<!ELEMENT zipcode (#PCDATA)>
<!ELEMENT segments (segment*)>
<!ELEMENT segment (start, end, name, leftzip, rightzip, leftsns,
rightsns, streetclass, increasingAddresses)>
<!ELEMENT start (lat, long)>
<!ELEMENT end (lat, long)>
<!ELEMENT leftzip (#PCDATA)>
<!ELEMENT rightzip (#PCDATA)>
<!ELEMENT streetclass (#PCDATA)>
<!ELEMENT leftsns (#PCDATA)>
<!ELEMENT rightsns (#PCDATA)>
```

```
<!ELEMENT name (#PCDATA)>
<!ELEMENT increasingAddresses (#PCDATA)>

<!ELEMENT lat (#PCDATA)>
<!ELEMENT long (#PCDATA)>
```

SportsScores.dtd

```
<!ELEMENT sportsscores (module, score*)>

<!ELEMENT module (#PCDATA)>
<!ELEMENT score (id, team1, score1, team2, score2, periodname,
periodvalue)>

<!ELEMENT id (#PCDATA)>
<!ELEMENT team1 (#PCDATA)>
<!ELEMENT score1 (#PCDATA)>
<!ELEMENT team2 (#PCDATA)>
<!ELEMENT score2 (#PCDATA)>
<!ELEMENT periodname (#PCDATA)>
<!ELEMENT periodvalue (#PCDATA)>
```

TrafficReports.dtd

```
<!ELEMENT trafficreports (module, report*)>

<!ELEMENT module (#PCDATA)>
<!ELEMENT report (id, streetname, starttime, endtime, timestamp)>

<!ELEMENT id (#PCDATA)>
<!ELEMENT streetname (#PCDATA)>
<!ELEMENT starttime (#PCDATA)>
<!ELEMENT endtime (#PCDATA)>
<!ELEMENT timestamp (#PCDATA)>
```

Sample Data for Prototype

Cars (compay, car type, capacity):
Boston, limo, 10
Cambridge, sedan, 3
Boston, sedan, 4
Cambridge, limo, 8

Images (image name, location id, location name, filename):
Wang Center Stairs, 1, Wang Center, wang4.jpg
Wang Center Theater, 1, Wang Center, wang1.jpg
Wang Center Ceiling, 1, Wang Center, wang2.jpg
Front of Tang Hall, 2, Tang Hall, tang1.jpg
Back of Tang Hall, 2, Tang Hall, tang2.jpg
Hot Dog Stand, 4, Fenway Park, fenway1.jpg
View from Goodyear Blimp, 4, Fenway Park, fenway2.jpg
Front of John Harvard's, 6, John Harvard's, john1.jpg
Interior of John Harvard's, 6, John Harvard's, john2.jpg
Doorstep of Eliot Hotel, 7, Eliot Hotel, eliot1.jpg
Lobby of Eliot Hotel, 7, Eliot Hotel, eliot2.jpg
Side of Legal Seafood, 8, Legal Seafood, legal1.jpg
Back of Legal Seafood, 8, Legal Seafood, legal2.jpg
Lumiere Dining Room, 9, Lumiere, lumiere1.jpg
Lumiere Table, 9, Lumiere, lumiere2.jpg
First Floor of Ryles, 10, Ryles, ryles1.jpg
Second Floor of Ryles, 10, Ryles, ryles2.jpg
Front of Museum, 10, Museum of Fine Arts, museum1.jpg
Side of Brian's House, 10, Brian's House, brian1.jpg
John Harvard's Men's Room, 6, John Harvard's, john3.jpg
Customer at John Harvard's, 6, John Harvard's, john4.jpg
Wang Center Stage, 1, Wang Center, wang3.jpg
Field at Fenway Park, 4, Fenway Park, fenway3.jpg
Baseball Player at Fenway Park, 4, Fenway Park, fenway4.jpg
Lobby of Tang Hall, 2, Tang Hall, tang4.jpg
Brian's Room, 2, Brian's House, tang3.jpg
Bar at Legal Seafood, 8, Legal Seafood, legal3.jpg
Food at Legal Seafood, 8, Legal Seafood, legal4.jpg
Entrance of Ryles, 10, Ryles, ryles3.jpg
Dancers at Ryles, 10, Ryles, ryles4.jpg
Room at the Eliot Hotel, 7, Eliot Hotel, eliot3.jpg
Eliot Hotel Towel, 7, Eliot Hotel, eliot4.jpg
Chef at Lumiere, 9, Lumiere, lumiere3.jpg
Food at Lumiere, 9, Lumiere, lumiere4.jpg

Locations (name, id, street number, street name, zipcode, function code, capacity, hours, pickups points):

Wang Center, 15, Roberts Ln, 02554, 00001, 1300, 6:00 PM - 9:00 PM, Back, Front
Tang Hall, 77, Main St, 02554, 00002, 250, 8:00 AM - 10:00 PM, Back, Side
Museum of Fine Arts, 26, Sesachacha Rd, 02554, 00002, 250, 8:00 AM - 10:00 PM, Back, Side
Fenway Park, 5, McKinley Ave, 02564, 00002, 250, 8:00 AM - 10:00 PM, Back, Side
Brian's House, 80, Surfside Dr, 02554, 00002, 250, 8:00 AM - 10:00 PM, Back, Side
John Harvard's, 40, Millbrook Rd, 02554, 00002, 250, 8:00 AM - 10:00 PM, Back, Side
Eliot Hotel, 6, Stone Post Way, 02584, 00002, 250, 8:00 AM - 10:00 PM, Back, Side, Front
Legal Seafood, 25, Millbrook Rd, 02554, 00002, 250, 8:00 AM - 10:00 PM, Back, Side
Lumiere, 12, Pochick St, 02554, 00002, 250, 8:00 AM - 10:00 PM, Back, Side
Ryles, 11, Longwood Dr, 02564,00005, 200, 5:00 PM - 2:00 PM, Front
Starbucks, 72, Orange St, 02554, 00010, 75, 8:00 AM - 7:00 PM, Front, Side
Dunkin Donuts, 111, Somerset Rd, 02554, 50, 7:00 AM - 8:00 PM, Front
King School, 42, Wauwinet Rd, 02554, 850, 6:00 AM - 5:00 PM, Front, Side, Back

Pickups (pickup street number, pickup street name, pickup zip code, pickup location id, pickup location name, pickup pickup point, dropoff street number, dropoff street name, dropoff zip code, dropoff location id, dropoff location name, dropoff pickup point, time, customer type, number of passengers):

80, Surfside Dr, 02554, 5, Brian's House, Front, 26, Sesachacha Rd, 02554, 3, Museum of Fine Arts, Front, 12:00 PM, normal, 2
5, McKinley Ave, 02564, 4, Fenway Park, Side, 40, Millbrook Rd, 02554, 6, John Harvard's, Front, 5:00 PM, normal, 2
6, Stone Post Way, 02584, 7, Eliot Hotel, Front, 25, Millbrook Rd, 02554, 8, Legal Seafood, Front, 6:00 PM, business, 4
25, Millbrook Rd, 02554, 8, Legal Seafood, Front,6, Stone Post Way, 02584, 7, Eliot Hotel, Front,8:00 PM, business, 4
77, Main St, 02554,2, Tang Hall, Back,12, Pochick St, 02554,9,Lumiere,Front,5:00 PM, normal, 2
12, Pochick St, 02554,9,Lumiere,Front, 15, Roberts Ln, 02554, 1, Wang Center, Front, 7:00 PM, normal, 2
15, Roberts Ln, 02554, 1, Wang Center, Front, 11, Longwood Dr, 02564, 10, Ryles, Front, 10:00 PM, normal, 4
80, Surfside Dr, 02554, 5, Brian's House, Back, 111, Somerset Rd, 02554,12, Dunkin Donuts, Front , 9:00 AM, normal, 3
42, Wauwinet Rd, 02554, 13, King School, Front,72, Orange St, 02554,11,Starbucks,Side, 3:00 PM, normal, 5

Tuesday, 22 May 2001 22:35:38

Please reuse this banner page if you find it useful, or recycle it if you don't.

To find answers to common printing questions, type

```
olc answers
```

at the `athena%` prompt, and choose `"PRINTING Answers"`.

bapurvil

biohazard-cafe
w20thesis:thesisprn.prn